AN ANYTIME APPROACH TO CONNECTIONIST THEORY REFINEMENT: REFINING THE TOPOLOGIES OF KNOWLEDGE-BASED NEURAL NETWORKS

By

David William Opitz

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1995

To Mom, Dad, Trish, and Sarah
for having patience for all my foolishness.

Abstract

Many scientific and industrial problems can be better understood by learning from samples of the task at hand. For this reason, the machine learning and statistics communities devote considerable research effort on generating inductive-learning algorithms that try to learn the true "concept" of a task from a set of its examples. Often times, however, one has additional resources readily available, but largely unused, that can improve the concept that these learning algorithms generate. These resources include available computer cycles, as well as prior knowledge describing what is currently known about the domain. Effective utilization of available computer time is important since for most domains an expert is willing to wait for weeks, or even months, if a learning system can produce an improved concept. Using prior knowledge is important since it can contain information not present in the current set of training examples.

In this thesis, I present three "anytime" approaches to connectionist theory refinement. Briefly, these approaches start by translating a set of rules describing what is currently known about the domain into a neural network, thus generating a *knowledge-based* neural network (KNN). My approaches then utilize available computer time to improve this KNN by continually refining its weights and topology. My first method, TopGen, searches for good "local" refinements to the KNN topology. It does this by adding nodes to the KNN in a manner analogous to symbolically adding rules and conjuncts to an incorrect rule base. My next approach, Regent, uses genetic algorithms to find better "global" changes to this topology. Regent proceeds by using (a) the domain-specific rules to help create the initial population of KNNs and (b) crossover and mutation operators specifically designed for KNNs. My final algorithm, Addemup, searches for an "ensemble" of KNNs that work together to produce an effective composite prediction. Addemup works by using genetic algorithms to continually create new networks, keeping the set of networks that are as accurate as possible while disagreeing with each other as much as possible. Empirical results show that these algorithms successfully achieve each of their respective goals.

Acknowledgments

A thesis does not get written with only taking one casualty. No, instead the process of thinking up, investigating, and writing a thesis affects (and in many cases disturbs) the lives of many along the way. So, before I continue, I would like to give thanks to all those who's contributions made this thesis possible.

The greatest thanks for my thesis work goes to my advisor, Jude Shavlik, for his intellectual and financial support. He provided not only a plethora of ideas along the way, but helped refine them and repeatedly pointed me back in the right direction whenever I was sure I wanted to get lost. Certainly, most of the research skills I learned in graduate school I owe to Jude. Research grants that have supported me through graduate school include a WARF fellowship, an IBM grant, a NYNEX grant, DOE Grant DE-FG02-91ER61129, NSF Grant IRI-9002413, and finally ONR Grants N00014-90-J-1941 and N00014-93-1-0998.

Thanks to my committee members Olvi Mangasarian, Blake LeBaron, Charles Dyer, and Wei-Yin Loh for providing insightful comments that improved my thesis. Thanks also to my fellow MLRG research partners throughout the years. Mark Craven, Richard Maclin, Kevin Cherkauer, Nick Street, Carolyn Allex, and others provided some type of support and/or contribution along this journey. Finally, thanks to Mick Noordewier of Rutgers for creating and commenting on the DNA datasets and domain theories that I used in this thesis.

I would also like to thank all my family and friends for all their steadfast support, encouragement, and much needed distractions. Thanks to my sister Kim for always reminding me just how long I (we) have been in school. A special thanks to my parents for not only providing me with the drive and support needed to succeed, but for always being there for me no matter what silly path I chose to take. Especially though, for pointing out the finer things in life outside research, and not letting me take graduate school too seriously.

Finally, but certainly not least, I would like to thank my wife Trish for all her positive encouragement during the course of this long process. Without her, not only would this document be unfinished, I would probably be in a padded cell somewhere under state control. Her fending off the inevitable questions of "Are you *still* in school? When are you *finally* going to finish?" from friends and relatives during our visits home to Montana will forever be appreciated. As will all the rides to and from work during inclement weather (i.e., most of the time in Wisconsin) and "getting things done" so we could spend quality time together with Sarah. I only hope that I can provide her the same type of support and positive attitude for any venture she chooses to pursue.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Many scientific and industrial problems can be better understood by learning from samples of the task at hand. Thus the machine learning and statistics communities devote considerable research effort in generating inductive-learning algorithms that try to learn the true "concept" of a task from a set of its examples (Shavlik & Dietterich, 1990; Weiss & Kulikowski, 1990). Often, however, one has additional resources readily available that can improve the concept that these learning algorithms generate. These resources include available computer cycles, as well as prior knowledge describing what is currently known about the domain. The goal of my thesis revolves around developing "connectionist" learning systems that are able to effectively use available data, background knowledge, and computer cycles to generate the most accurate concept possible.

The following example illustrates the importance of this goal. Imagine you are a geneticist who is interested in finding where genes occur in DNA sequences. Laboratories throughout the world are producing large volumes of sequenced DNA data (Watson, 1990); however, direct laboratory analysis of this data is difficult and expensive, and you are therefore finding it impossible to keep up. To help you with this predicament, you attempt to create an algorithm that can simply look at a DNA sequence and find, with high certainty, where genes are likely to occur in the sequence.

To generate such an algorithm, you might first try gathering examples of gene and non-gene sequences from previously studied DNA. You could then apply an inductive-learning algorithm (e.g., backpropagation, Rumelhart et al., 1986, or C4.5, Quinlan, 1993) on these examples, producing a concept that predicts which ones encode genes. You would quickly realize that given both the complexity of the problem and the limited amount of data, the quality of the concepts that typical inductive learners generate are not correct enough to be fully helpful.

Not to be deterred, you also realize that in addition to the set of data, you have an expert's knowledge of what is currently believed to be the important subsequences found in most genes. You may have tried directly encoding this knowledge in the past, as a traditional program; however, this knowledge is just not yet correct enough to be useful by itself. Nonetheless, this partially correct knowledge is a good first approximation and, as will be seen, can in many cases improve the concept produced by inductive-learning systems.

By priming the inductive-learning algorithm with your current knowledge, you would find that you obtained a better concept than before, though still not as good as you desire. One thing you notice however, is that the learning systems usually generate one solution (i.e., one concept) then stop. This is disturbing to you, since you work for one of the larger labs in the world, and have access to a tremendous amount of computer power. Not only are there a lot of computer cycles available on your site, but your lab has access to many other computers throughout the world as well. What you desire then, is an algorithm that can continually improve the quality of its solution as a function of available computer time. After all, not only would an improved solution make your job more productive, it would also produce a better understanding of the function of DNA, which is useful from a scientific point of view,

My thesis focus is to produce learning algorithms that are able to capitalize on all available resources, not just training data, when inducing their concepts. The particular type of learning algorithms I focus on are feed-forward neural networks (Hertz et al., 1991). Neural networks are a general approach that has been applied to a wide variety of real-world problems (Sejnowski & Rosenberg, 1987; Uberbacher & Mural, 1991; Pomerleau, 1991; Rost & Sander, 1993) and empirical studies have shown that neural networks generalize to novel examples as well as, or in some cases better than, many other common learning algorithms (Atlas et al., 1989; Fisher & McKusick, 1989; Mooney et al., 1989; Tsoi & Pearson, 1990). Before giving my thesis statement and general framework for my algorithms, however, I present a brief overview of learning from training data, using background knowledge, and exploiting available computer power.

1.1 Using Training Data: Inductive Learning

A system that learns from a set of labeled examples is called an *inductive learner* (alternately, a *supervised*, *empirical*, or *similarity-based* learner). The output for each labeled example is provided by a teacher, and the set of examples given to a learner is called the *training* set. The task of inductive learning is to generate from the training set a concept description that correctly predicts the output of all future examples, not just those from the training set. (I henceforth use the term *generalization* to mean classification accuracy on examples not seen during training).

This type of learning encompasses many applications, such as our previously described task of determining if a gene lies within a particular DNA sequence. In this case, the representation of each example is the "window" of DNA under consideration. Real-valued inputs, such as ones that represent the three-dimensional structure of the sequence, may also be given. The output in this case is whether or not a gene occurs in this region of DNA.

Several inductive-learning algorithms have been previously studied (Mitchell, 1982; Breiman et al., 1984; Rumelhart et al., 1986; Quinlan, 1986). These algorithms differ both in their concept-representation language, and in their method (or *bias*) of constructing a concept within this language. These differences are important since they determine which concepts a classifier will induce. Experimental methods, based on setting aside a "test set" of instances, judge the generalization performance of the inductive learner (Weiss & Kulikowski, 1990, Chapter2). The instances in the test set are not used during the training process, but only to estimate the learner's predictive accuracy. For further reading, see Michalski (1983), Shavlik and Dietterich (1990), or Weiss and Kulikowski (1990).

1.2 Using Background Knowledge: Theory Refinement

An alternative to the inductive learning paradigm is to build a concept not from a set of examples, but by querying experts in the field and directly assembling a set of rules that solve the task (i.e., build an *expert system*, Waterman, 1986). A problem with building expert systems is that the rules derived from the experts (which, following the convention in the machine learning literature, I refer to as a *domain theory* from now on) tend to be only approximately correct. Thus, while the domain theory is usually a good first approximation of the concept to be learned, inaccuracies are frequently exposed during empirical testing. *Theory-refinement systems* (Ginsberg, 1990; Pazzani & Kibler, 1992; Ourston & Mooney, 1994; Towell & Shavlik, 1994) are systems that go about revising a domain theory on the basis of a collection of examples. These systems try to improve the theory by making repairs that minimize changes to the theory, while making it consistent with the training data. Changes to the initial domain theory should be kept to a minimum because the domain theory presumably contains useful information, even if it is not completely correct.

These hybrid learning systems are designed to learn from both theory and data, and empirical tests have shown them to achieve high generalization with significantly fewer examples than purely inductive-learning techniques (Pazzani & Kibler, 1992; Ourston & Mooney, 1994; Towell & Shavlik, 1994). Thus an ideal inductive-learning system must be able to incorporate any background knowledge that is available in the form of a domain theory to improve its ability to generalize.

Several theory-refinement systems use neural networks as their inductive-learning component. These knowledge-based connectionist approaches have been shown to frequently generalize better than many other machine learning systems (Fu, 1989; Towell, 1991; Tresp et al., 1992; Lacher et al., 1992; Mahoney & Mooney, 1994). These systems proceed by first translating the domain theory directly into a neural network, thereby determining the network's topology and initial weight settings. They then refine these reformulated rules using standard neural-learning techniques such as backpropagation (Rumelhart et al., 1986).

One of the most successful of these approaches is the KBANN system (Towell, 1991; Towell & Shavlik, 1994). KBANN is designed for domain theories represented by Prolog-style, propositional rules. KBANN, and other connectionist theory-refinement systems that do not alter their network topologies, suffer when given *impoverished* domain theories – ones that are missing rules needed to adequately learn the true concept (Opitz & Shavlik, 1993; Towell & Shavlik, 1994). I cover KBANN and the other theory-refinement systems in more detail in Section 2.2.1. *The main focus of the new algorithms I present in Chapters 3, 4, and 5 is how to refine the topology of the knowledge-based neural networks produced by* KBANN.

1.3 Using Available Computer Power: Anytime Learning

As was the case with our hypothetical geneticist above, many domain experts are willing to wait for weeks, or even months, if a learning system can produce an improved theory. This, coupled with the fact that computing power is rapidly growing, illustrates the importance of developing "anytime" learners that are able to trade off the expense of large numbers of computing cycles for gains in predictive accuracy. Dean and Boddy (1988) defined the criteria for an anytime algorithm to be: (a) the algorithm can be suspended and then resumed with minimal overhead, (b) the algorithm can be stopped at any time and return an answer, and (c) the algorithm must return answers that improve over time. While these criteria were created for planning and scheduling algorithms, I believe machine learning researchers should create anytime algorithms for inductive learning as well.[1] Such learning algorithms should produce a good concept quickly, then continue to search the space of available concepts, reporting the new "best" concept whenever one is found.

Most standard inductive learners such as backpropagation (Rumelhart et al., 1986) and ID3 (Quinlan, 1986), however, are unable to continually improve their answers (at least until they receive additional training examples). In fact, if run too long, these algorithms tend to "overfit" the training set (Holder, 1991). Overfitting occurs when the learning algorithm produces a concept that captures too much information about the training examples, and not enough about the general characteristics of the domain as a whole. While these concepts do a great job of classifying the training instances, they do a poor job of generalizing to new examples – our ultimate measure of success. To help illustrate this point, consider the typical regression case shown in Figure 1. Here, fitting noisy data with a high-degree polynomial is likely to lead to poor generalization. To avoid overfitting, many algorithms employ some form of Occam's Razor (Blumer et al., 1987) by preferring simple descriptions over complex ones, even if the simple one does not fit the training set as accurately.

The framework I use for making my algorithms anytime learners is quite simple. I spend my computer time considering many different possible concept descriptions, scoring each possibility, and always keeping the description (or set of descriptions) that score best. My framework is *anytime* with respect to the scoring function. Assuming the scoring function is accurate, then as long as I am considering a wide range of "good" possibilities, the quality of my final concept should continually improve; however, since the scoring function is only an *approximate* measure of generalization, there is no guarantee that

[1] My use of the term *anytime learning* differs from that of Grefenstette and Ramsey (1992); they use it to mean continuous learning in a changing environment.

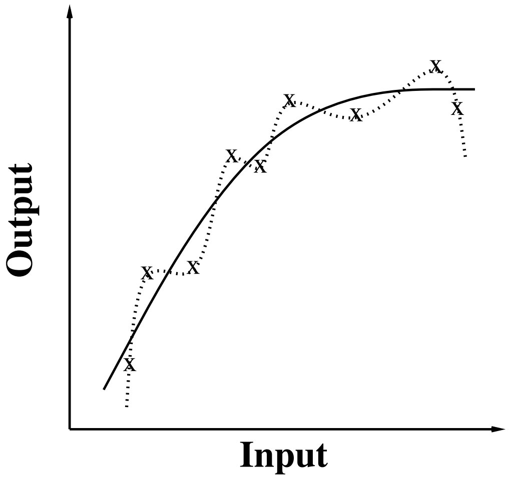


Figure 1: This is a classical regression example where a smooth function (the solid curve) that does not fit all of the noisy data points (the x's) is probably a better predictor than a high-degree, polynomial (the dashed curve).

generalization will not decrease over time. The detailed description of how I apply this framework to neural networks is presented next.

1.4 Thesis Statement

As motivated in the previous three sections, an ideal inductive-learning algorithm should be able to prime its learning with what is currently known about the domain, and then be able to continually improve its concept description over time. Figure 2 illustrates the framework of the three new learning algorithms I introduce in this thesis.

My algorithms start by having the KBANN algorithm translate the domain theory directly into a neural network. Given the proven effectiveness of this method, this gives a good initial guess for an appropriate network topology and weight settings. I then train this network with standard learning techniques (Rumelhart et al., 1986) and score it according to its estimated generalization ability. I set aside a *validation* set[2] for use

[2]A *validation* set is a subset of the training instances that is set aside before training. The validation set is not seen by the learning algorithm during training, but is instead used to judge generalization performance after training. The validation set is distinct from a *test* set often used by machine learning

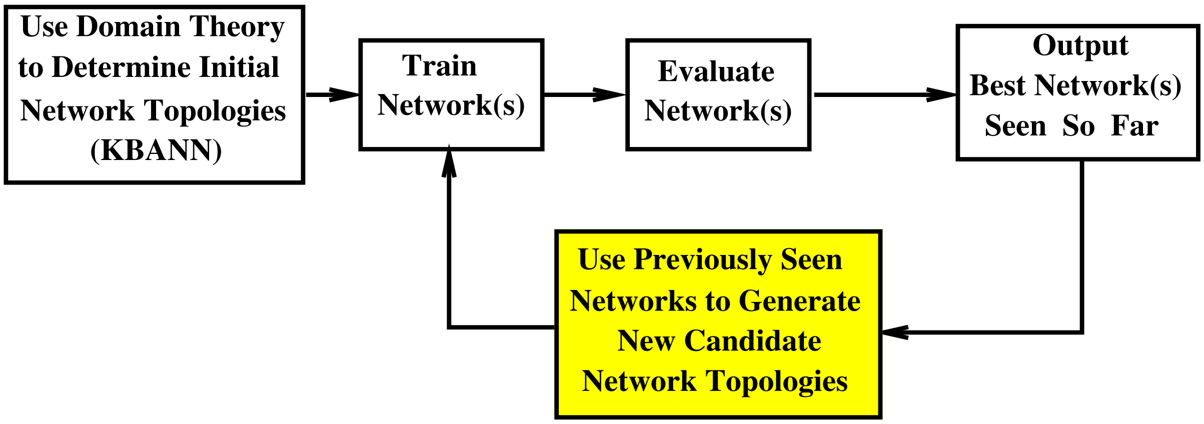


Figure 2: Framework for the learning algorithms I introduce in this thesis.

---

by my scoring function (future scoring functions I plan to investigate are discussed in Section 7.2.1). By using neural networks, my algorithms are *incremental* in that they can revise their existing concept as new data arrives, rather than having to restart training from the beginning.

As stated above, while KBANN gives a good initial guess for an appropriate network topology, this network needs to be refined to ensure appropriate corrections to the domain theory can be made. My algorithms continually create new networks by altering or combining previously trained networks. The current best network (or set of networks) is always output to allow an anytime approach to learning. While many authors have addressed creating and training knowledge-based neural networks (KNNs), few have addressed the highlighted portion of Figure 2 – how should we best search the space of possible KNN topologies. *This is the major focus of this thesis.*

The three algorithms I present in this thesis are created with this framework in mind and mainly differ in their approach of searching through the space of possible KNN topologies. My first algorithm, TopGen (Topology Generator), modifies KBANN's topology by continually adding new nodes to the network in a manner analogous to adding rules and

researchers; the validation set is part of the training set.

conjuncts to a symbolic rule base. My second approach, Regent (REfining, with Genetic Evolution, Network Topologies), considers a broader range of network topologies than TopGen. It does this by translating the domain theory into a "population" of networks, then uses genetic algorithms (Holland, 1975; Goldberg, 1989) to continually create new networks. My final algorithm, Addemup (Accurate anD Diverse Ensemble-Maker giving United Predictions), searches for a set of accurate and diverse KNNs, to produce an overall prediction that is the weighted combination of the prediction of each network in the set. All of these algorithms address the following, which is the main statement made by this dissertation:

Thesis: *An effective learning system must be able to take advantage of all available resources to improve the quality of the concept it generates. These resources include training data, available computer time, and background knowledge describing what is currently known about the domain. An ideal learning system, then, should be able prime its learning with the background rules, and be able to continually improve its concept over time using the set of training data.*

1.5 Thesis Overview

The rest of this dissertation motivates, describes, and empirically tests the three new connectionist learning systems (TopGen, Regent, and Addemup) that I propose in this thesis. The remaining chapters are organized as follows:

- In Chapter 2, I start by giving an overview of feed-forward neural networks. In particular, I focus on current techniques for training these networks, as well as the importance of finding a good topology. This is followed by an explanation of how background knowledge can be incorporated into neural networks. Specifically, I concentrate on the Kbann system, describing its strengths and weaknesses.

- I present my first approach, TopGen, in Chapter 3. TopGen heuristically searches the possible expansions of a KNN, guided by the domain theory, the network, and the training data. It does this by dynamically adding hidden nodes to the neural representation of the domain theory in a manner analogous to adding rules and conjuncts to the domain theory. Experiments indicate that TopGen is able to heuristically find effective places to add nodes to the KNN and verify that new nodes must be added in an intelligent manner.

- Chapter 4 presents my second approach, Regent, which uses genetic algorithms to broaden the type of networks seen during its search. It does this by using (a) the domain theory to help create an initial population and (b) crossover and mutation operators specifically designed for KNNs. Experiments indicate that Regent is able to significantly increase generalization compared to Kbann and TopGen.

- I present my last approach, Addemup, in Chapter 5. This algorithm works by using genetic algorithms to directly search for an accurate and diverse set of trained KNNs to be used in a neural-network *ensemble* – an ensemble is a set of separately trained networks whose predictions are combined through some weighting scheme; such approaches have proven effective at substantially increasing generalization (Hansen & Salamon, 1990; Perrone, 1992; Wolpert, 1992). Experiments show that Addemup is able to effectively incorporate a domain theory to help generate a set of networks that are more accurate than several existing approaches.

- Chapter 6 reviews additional related work not covered in previous chapters, while Chapter 7 presents a summary of the contributions of my work, as well as limitations and future research to the framework of this thesis as a whole. Limitations and future work specific to each learning algorithm are presented at the end of that algorithm's chapter.

- Appendix A presents the problem domains that I used to test my algorithms. These domains consist of an artificial chess-related domain, four real-world Human-Genome Project domains, and finally another real-world domain, from Nynex Corporation, for diagnosing faults in a telephone loop.

Chapter 2

What is a Knowledge-Based Neural Network?

The learning method I concentrate on in this thesis is that of neural networks (Hertz et al., 1991). Specifically, my algorithms translate a given domain theory directly into a neural network, then continually improve this *knowledge-based network* over time. Before presenting my three approaches, I give a high-level overview of neural networks, including what they are, how they are trained, and important issues pertaining to their generalization performance. I then give a brief history of "connectionist" theory-refinement systems. In particular, I concentrate on the KBANN system (Towell, 1991; Towell & Shavlik, 1994), giving an brief overview of this algorithm and its strengths, before illustrating one of its weaknesses.

2.1 Introduction to Artificial Neural Networks

A neural network consists of nodes interconnected by weighted links. In this thesis, I focus only on *feed-forward* neural networks – networks that propagate a set of input signals, representing an example's feature values, forward into a set of output signals that serve as the network's prediction. Figure 3 shows an example of such a network. The

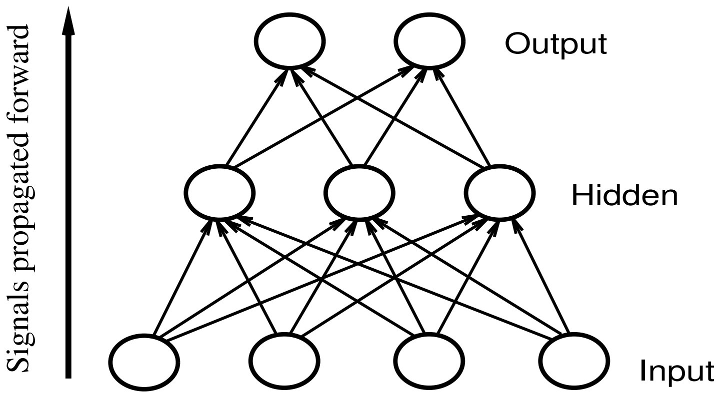


Figure 3: A "standard" feed-forward neural network.

network's connectivity may be arbitrary and there need not be the notion of a "layered" network, but there cannot be cycles in a feed-forward network. Nodes that are neither input nor output nodes are called hidden nodes (since they are not seen by the outside environment); their sole function is to help map input values to output values.

Signals are propagated forward through a network by multiplying the output of a node (called the node's *activation*) by the weight of outgoing links. These signals serve as part of the input to recipient nodes; more specifically, the input of a node is the sum of the incoming signals plus a learnable bias term:

$$net_i = \sum_j w_{ij} a_j + \theta_i \ , \tag{1}$$

where net_i is the input to node i, w_{ij} is the arc weight from node j to node i, a_j is the activation of node j, and θ_i is the "bias" term for node i. The bias is similar in function to a threshold, and can be learned just like any other weight. Each non-input node maps its real-valued input (net_i) to a real-valued activation (a_i) using a function called the *activation function*. Most network training techniques require that the derivative of the activation function exist. Also, this activation function cannot be just a linear function, since this type of function can only model "linear-separable" concepts.[1] The most commonly used nonlinear and differentiable activation function is

[1] A *linear-separable* concept in an n dimensional input space is one where a single $n-1$ dimensional hyperplane can separate the positive instances from the negative instances.

the sigmoid function:

$$a_i = \frac{1}{1 + e^{-net_i}}. \tag{2}$$

This activation function "squashes" a node's input to an activation value that is between 0 and 1.

Once we decide on an activation function, the topology of a network (i.e., the connectivity of the nodes) and the weight values on these connections determine the function of the network. Many authors have developed learning rules for finding an appropriate set of weight values for a fixed topology. While I present a brief overview of the most relevant of these learning rules next, one may refer to Rumelhart et al. (1986), Hertz et al. (1991), or Rumelhart et al. (1995) for further reading.

2.1.1 Training a Neural Network

Most neural-network learning rules are gradient methods that work by first propagating a training example's inputs forward through the network, then comparing the activation of the output nodes with the corresponding "target" values of the training example. The error between the two is then propagated back through the network, and the weights in the network are changed to reduce this error. Figure 4 shows an example of this process.

The most commonly used learning rule is *backpropagation*, which is the one I use in this thesis. It has been discovered, then re-discovered, several times (Bryson & Ho, 1969; Werbos, 1974; Rumelhart et al., 1986). Backpropagation changes weights in the network according to the derivative of an "error function," E, with respect to each weight:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_i} \times \frac{\partial net_i}{\partial w_{ij}} = \left(\frac{\partial E}{\partial a_i} \times \frac{\partial a_i}{\partial net_i} \right) \times \frac{\partial net_i}{\partial w_{ij}} \tag{3}$$

The first term, $\partial E / \partial a_i$, depends of the error function E (explained below) and can be directly calculated for output nodes; however, since there is no specific target activation for hidden nodes, the error for these nodes must be determined recursively in terms of the error of the nodes to which it directly connects ($\partial E / \partial net_i$) and the weights of those connections. Thus this error is "back propagated" in the following fashion for hidden

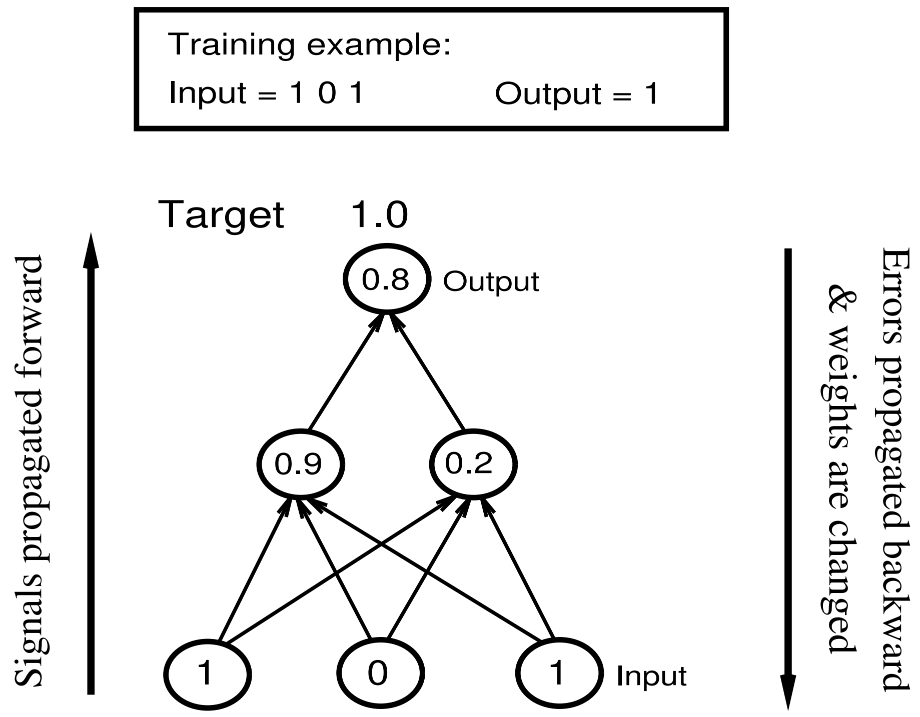


Figure 4: Training a neural network. The network's input nodes are set to the corresponding values of the training instance (1, 0, 1 in this case). These values are propagated forward through the network, creating an activation of 0.8 at the output node. This activation is compared to the target value of 1, and the error between these two is propagated back through the network. Finally, the weights in the network are changed to reduce this error.

node i:

$$\frac{\partial E}{\partial a_i} = \sum_k \left[\left(\frac{\partial E}{\partial a_k} \times \frac{\partial a_k}{\partial net_k} \right) \times w_{ki} \right]. \tag{4}$$

Because we wish to reduce error, the magnitude of the actual weight change is:

$$\Delta w_{ij}(t) = -\eta \times \frac{\partial E}{\partial w_{ij}} + \alpha \times \Delta w_{ij}(t-1), \tag{5}$$

where the scalar η is the learning rate, and $\alpha \times \Delta w_{ij}(t-1)$ is the *momentum* term (Rumelhart et al., 1986). The momentum term, similar to that of conjugate gradient's (Fletcher, 1987, Chapter 4), helps prevent the updates from oscillating wildly, encouraging changes that are made in the direction of the *average* downhill force. The momentum parameter, α, should be between 0 and 1; I use the commonly chosen value of 0.9.

One can view backpropagation as a *gradient* method in "weight" space – the space spanned by the free parameters in the networks, namely its weights and biases. In *online* learning, the error for a single training example is decreased at each step; however, there is no guarantee that the error of the overall objective function on *all* training examples also decreases during this step. Thus online backpropagation is not true gradient descent, rather it should be viewed as a *nonmonotone perturbed gradient* algorithm (Mangasarian & Solodov, 1994). *Batch* learning, on the other hand, only updates the weights after all training examples have been presented and can thus be viewed as pure gradient descent; however, online learning has proven to be a more accurate training method (LeCun & Bengio, 1995). Also, Mangasarian and Solodov (1994) proved that, under certain natural assumptions, online backpropagation converges.

Error Functions

Naturally, the $\partial E/\partial a_i$ term for the output nodes depends on the error function, E, being minimized. Our task now is to find the appropriate error function for the problem we are trying to solve. To develop the basis for an appropriate error function, I follow the discussion in Rumelhart et al. (1995). I start by assuming that our goal is to find the weights that are the most probable explanation of our data. One can express this goal in the form of Bayes' Theorem:

$$P(N|D) = \frac{P(D|N)P(N)}{P(D)}, \tag{6}$$

where D represents the observed data, and N represents the network with specified weights and biases. We then take the log of this probability, since sums are easier to work with than products and we can maximize a probability by maximizing its log. Taking the log of Equation 6 produces:

$$ln\ P(N|D) = ln\ P(D|N) + ln\ P(N) - ln\ P(D). \tag{7}$$

The last term can be disregarded since it is independent of the network and all we are concerned with is the most probable network. The first term, the accuracy term, measures how well the network accounts for the data. The second term is a measure of the prior probability of the network. It is common to concentrate only on the first term; however, I will return to the importance of the second term later.

To maximize $lnP(D|N)$, we need to know the distribution that the network's outputs are to learn. Table 1 shows three probability distributions that are commonly assumed in practice, and their appropriate error functions (Rumelhart et al., 1995). The first column, the Gaussian case, assumes that the network models a regression task and that the noise is normally distributed about the predicted values. The negative of the log of this probability gives us an error function, E, that is proportional to squared error. This is backpropagation's original, and still most common, error function. The activation function of the output nodes in this case was originally assumed to be the sigmoidal function (Rumelhart et al., 1986); however, given the Guassian assumption of error, Rumelhart et al. (1995) show it makes more sense to have the *output* node's activation be a linear function of its net input. (Note the activation function for the hidden nodes is still commonly the sigmoidal function.) Making this assumption gives us the simple update rule, $\partial E/\partial net_i = t_i - a_i$.

Another common application for networks is to learn a binary output for each input

Table 1: Error functions for back propagation. This table, derived from Rumelhart et al. (1995), gives the appropriate error function (E), activation function (a_i) for the output nodes, and two relevant derivatives from Equation 3 for three commonly assumed probability distributions. All summations range over all the output nodes.

	Gaussian	Binomial	one-of-N
E	$\frac{1}{2}\sum(t_i - a_i)^2$	$-\sum[t_i\ ln(a_i) + (1-t_i)\ ln(1-a_i)]$	$-\sum t_i\ ln(a_i)$
a_i	net_i	$\frac{1}{1+e^{-net_i}}$	$\frac{e^{net_i}}{\sum e^{net_j}}$
$-\frac{\partial E}{\partial a_i}$	$t_i - a_i$	$\frac{t_i - a_i}{a_i(1-a_i)}$	$\frac{t_i}{a_i}$
$-\frac{\partial E}{\partial net_i}$	$t_i - a_i$	$t_i - a_i$	$t_i - a_i$

vector. The appropriate interpretation of the error in this case is the binomial distribution, which leads to the cross-entropy error function. Notice that choosing the sigmoid activation function with this interpretation also has the appealing result that the error for an output node ($\partial E / \partial net_i$) is simply $t_i - a_i$.

A widely used specialization of the binomial is the "one-of-N" classification task. In this case, the output for the correct class is a one, while all other outputs are zero. Rumelhart et al. (1995) suggest choosing the activation for the output nodes to be the normalized exponential:

$$a_i = \frac{e^{net_i}}{\sum_j e^{net_j}}. \tag{8}$$

They show that this activation is the generalization of the sigmoid (Equation 2) to the one-of-N task. By carefully choosing the normalized exponential to be the activation function of our output nodes, the error for the output node whose value should be 1 is again simply $t_i - a_i$. Towell (1991) found that picking the appropriate error and activation function is especially important with "knowledge-based" neural network (described later in this chapter).

Avoiding Overfitting

Generally, with each pass that our learning algorithm takes through the training set, the more accurately our network classifies these training instances. As with most inductive-learning algorithms (see Section 1.1), neural networks suffer from the problem of "overfitting" if the network learns the training instances too precisely (Holder, 1991; Weigend, 1993). In this case, the network does a good job of learning the training data, but does poorly on yet unseen cases; we do not consider a network to have truly learned a function until it performs well on these unseen instances. Combating the problem of overfitting has been the focus of many researchers.

One simple approach to prevent overfitting is to stop training before overfitting has occurred (Holder, 1991; Weigend et al., 1990). Although neural networks tend to have a large number of potential parameters, Weigend (1993) showed the various weights and hidden nodes in a network try to minimize the same error early during training, and are thus mostly functional duplicates of each other. Only as training progresses do the nodes start to diverge in their function. Therefore, the *effective* number of parameters is initially small, then increases during training. In fact, if training is stopped at the appropriate time, large networks tend to do as well as small, even "optimally" sized, networks (Weigend, 1993). The winner on one dataset at *The Santa Fe Time Series Prediction and Analysis Competition* (Weigend & Gershenfeld, 1992), for instance, had a network with more potential parameters than training instances (Wan, 1993).

One approach to stop training at the right moment is to use a validation set (Weigend et al., 1990). This set is part of the training instances, but is not used during the training of the network. Instead, one uses the validation set to estimate the performance of the network, and training is stopped when the performance of the validation set starts to decrease. Drawbacks of using a validation set are that the size of the training set is decreased; its effectiveness can be sensitive to the particular stopping criteria used

(Weigend et al., 1990); and it can be a noisy estimator of future performance (MacKay, 1992).

Above, I focused on the accuracy term, $lnP(D|N)$ of Equation 7; I now switch to the prior-probability term, $lnP(N)$, while again following the discussion in Rumelhart et al. (1995). We can potentially avoid overfitting by selecting an appropriate prior. In this case, we modify the weights of the networks with respect to the derivatives of both terms, not just the accuracy term. Therefore, the new weight-update rule is:

$$\Delta w_{ij}(t) = \eta \times \frac{\partial(lnP(D|N))}{\partial w_{ij}} + \gamma \times \frac{\partial(lnP(N))}{\partial w_{ij}} + \alpha \times \Delta w_{ij}(t-1), \tag{9}$$

where γ is the decay rate for the prior term. (I set γ to be $0.01 \times \eta$ in this thesis.) Since previously $E = -lnP(D|N)$, this update rule is the same as Equation 5, except for the added prior term.

Perhaps the most common prior term takes the form of *weight decay* (Hinton, 1986). If we assume our weights are independently drawn from a normal distribution about zero, we can write this prior-probability distribution as:

$$P(N) = exp\left(-\frac{\sum_{ij} w_{ij}^2}{2\sigma^2}\right). \tag{10}$$

Taking the derivative of the log of $P(N)$ we get:

$$\frac{\partial(lnP(N))}{\partial w_{ij}} = -\frac{w_{ij}}{\sigma^2}\ . \tag{11}$$

What this term does, then, is penalize large weights. The most likely network will be the one with the smallest weights that best classifies the data.

This decay term suffers, however, in that it prefers two smaller weights over one large weight (Nowlan & Hinton, 1992). For example, if a node receives input from two highly correlated nodes, it would prefer two connections with weights $(w/2)$ over the similarly behaved weights of w and 0 (since $(w/2)^2 + (w/2)^2 < w^2 + 0^2$). Thus this prior does not *eliminate* weights, as we would prefer following Occam's Razor. Instead of drawing weights from a single distribution about zero, a similar idea is to draw them from either a uniform distribution between $\pm\sigma_2$ or a normal distribution about zero. This has the effect of eliminating small weights by decaying them toward zero, while mostly leaving large weights alone. Weigend et al. (1990) express this distribution as:

$$P(N) = exp\left[-\sum_{ij} \frac{(w_{ij}/\sigma_1^2)^2}{1+(w_{ij}/\sigma_2^2)^2}\right] . \tag{12}$$

Taking the derivative of the log of $P(N)$ we get:

$$\frac{\partial\ (ln\ P(N))}{\partial\ w_{ij}} = -\frac{\sigma_2^2}{\sigma_1^2}\frac{w_{ij}}{(\sigma_2^2+w_{ij}^2)^2}. \tag{13}$$

I use this form of weight decay in this thesis; however, I decay weights back to their *initial* value, rather than toward zero (see Section 3.1.3). I do this since some of the weights correspond to rules in a domain theory (as explained later in this chapter), and decaying weights toward their initial value helps prevent unnecessary changes to the domain theory (Shavlik, 1994).

2.1.2 Finding an Appropriate Network Topology

While Weigend (1993) showed that large networks with many parameters can generalize as well as small networks if training is stopped in time, his experiments only involved standard single-hidden-layer networks. In general, the generalization ability of a network depends on finding a good, domain-dependent topology (Baum & Haussler, 1989; Tishby et al., 1989) and, since I use the gradient-descent method of backpropagation learning, a good set of initial weight settings.

One approach for finding an appropriate network topology, called *network-growing* algorithms, attempts to start with a small network and gradually grow it to the appropriate size. A number of researchers (Ash, 1989; Hanson, 1989; Wynne-Jones, 1991) have investigated networks that add new nodes within a single hidden layer in the course of learning. Although a single layer of hidden nodes can approximate any Boolean as well as any continuous function, they are incapable of creating higher-order features that may aid learning (Hertz et al., 1991). Also, as indicated above, the number of hidden nodes in single-hidden-layer networks does not matter as much as does finding a good point to stop training.

There have also been attempts to build *multi-layered* networks while learning progresses. Frean (1990) proposed the *upstart* algorithm, which adds two nodes to each node that has a high estimated error, one designed to correct false-positives and one designed to correct false-negatives. Mezard and Nadal (1989) proposed a *tiling* algorithm that starts from the bottom and works upwards, with each successive layer correcting the errors of the previous layer. The *cascade-correlation* algorithm (Fahlman & Lebiere, 1989) builds a hierarchy of hidden nodes in a cascaded manner, where each new node receives activation from all input nodes and all previously added hidden nodes.

Another approach is to start with a large network, then prune unimportant connections during training. Weight-pruning techniques differ from weight decay in that they initially train the network, remove unimportant weights, then train further this reduced network. This train-prune cycle repeats until training-set error starts to increase. A naive method to pruning is to assume low-weighted links are unimportant. Le Cun et al. (1989) and Hassibi and Stork (1992), however, use second-order gradient information to estimate the importance of each weight. Other methods (Hanson & Pratt, 1988; Chauvin, 1988; Mozer & Smolensky, 1989) try to remove unimportant *nodes* and their associated weights.

A final approach to finding an appropriate topology is to mount a "richer" search than hillclimbing through the space of topologies. This requires a method for searching through the topology space, as well as a function that can evaluate each topology. Such a search can be done using genetic algorithms (Goldberg, 1989; Harp et al., 1989; Miller et al., 1989; Oliker et al., 1992). In Chapter 4, I propose a method for using a genetic algorithm to refine the topology of *knowledge-based neural networks.* I introduce these types of neural networks next.

2.2 Overview of Knowledge-Based Neural Networks

Many researchers have proposed architectures for combining both symbolic and neural approaches to artificial intelligence. Neural networks have many advantages that make them attractive for complex, imprecise, and noisy problems – they generalize well to novel examples, are applicable to a wide variety of real-world problems, have high noise tolerance, and degrade gracefully in proportion to system damage. Symbolic methods, on the other hand, have advantages that make them attractive to problems that require explanation and understanding – their rules are generally intelligible to a human, background knowledge can be encoded in an easily manipulative form, and they allow easy control of information flow (Bookman & Sun, 1993).

Both neural and symbolic techniques have their disadvantages as well. Neural networks usually produce concepts that are hard to understand, their generalization performance can depend on finding a good domain-specific topology and initial weight settings, and training the network is a difficult task that can be time consuming. Alternately, symbol-processing systems usually generalize poorly on complex, imprecise problems, do not handle noisy data well, and do not gracefully degrade with a slight change of concept (Bookman & Sun, 1993).

An architecture that can incorporate both symbolic and neural methods can potentially produce a better system than either individually (Fu, 1989; Towell, 1991; Lacher et al., 1992; Mahoney & Mooney, 1994). Ideally, such a system should have the best of both worlds. It would be applicable to a wide variety of complex and noisy, real-world problems, and should still be able to naturally incorporate prior expert knowledge, while producing human-understandable concepts. There are many proposed methods for combining symbolic and connectionist learning; however, in this thesis I concentrate only on the popular approach shown in Figure 5 (Shavlik, 1994). The three steps in this framework are somewhat independent, and various portions have been addressed by several researchers; I focus on the largely ignored, but important, second step of refining a "knowledge-based" neural network's topology.

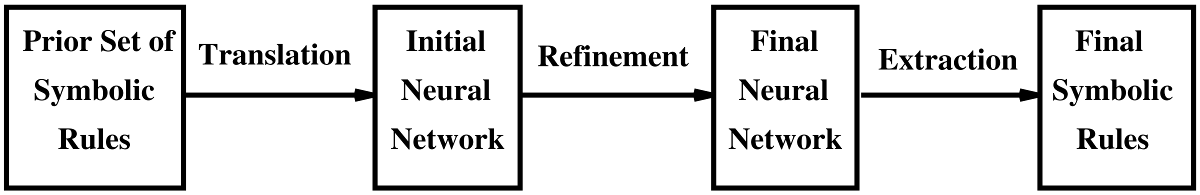


Figure 5: One approach for combining symbolic and neural learning.

As in my framework presented in Section 1.4, Figure 5's framework starts by translating the domain theory, represented by a set of symbolic rules, directly into a neural network. This translation process, which is explained in the next section, determines the initial weight settings and topology of the network. Hence these learners address the hard problem of finding a good domain-specific topology and initial weight settings by using prior knowledge of what is currently known about the domain. The resulting network is called a *knowledge-based neural network* (KNN).

These systems next refine the weights of the KNN by using standard neural-learning techniques. This type of refinement has been shown to be successful for a wide variety of domains; however, it is limited in the types of refinements that it can make. As I will illustrate in the next section, training the network without dynamically refining its topology does not allow the network to introduce new rules to the symbolic rule base. It only allows the re-weighting of the antecedents of existing rules. *Being able to dynamically refine the topology of a KNN is a major focus of this thesis.*

While the KNNs increase in generalization over standard neural networks is useful in and of itself, it is still desirable to understand what it is that the network has learned. The last step in this framework, then, is to extract humanly understandable rules that describe the function of the network. Several researchers have proposed methods for extracting rules from standard neural networks (Sestito & Dillon, 1990; Fu, 1991; Craven & Shavlik, 1994b). While this is a difficult problem, it is less daunting with KNNs since they usually obey two assumptions that allow them to be interpretable: (a) the meaning of its nodes does not significantly shift during training, and (b) almost all the nodes are either fully active or inactive. In fact, Towell and Shavlik (1993) presented an algorithm, called NofM, that is effectively able to extract rules from KNNs.

Many systems, designed for different types of rule bases, have been successfully implemented within this framework. For instance, systems have been designed for Prolog-style proposition rules (Towell & Shavlik, 1994), probabilistic rules (Fu, 1989; Mahoney & Mooney, 1994), finite-state grammars (Omlin & Giles, 1992; Maclin & Shavlik, 1993), mathematical equations (Roscheisen et al., 1991; Scott et al., 1992), statements in an imperative programming language (Maclin & Shavlik, 1994), and finally fuzzy-logic rules (Berenji, 1991; Masuoka et al., 1990). As stated before, many of these approaches have been shown to generalize better than other learning systems. The reason for much of this success has been attributed to the domain theory (a) suggesting potentially useful intermediate terms and (b) focusing attention on relevant inputs (Shavlik, 1994). Given the good initial weight settings, KNNs have also been shown to train faster than standard neural networks (Oliver & Schneider, 1988; Shavlik & Towell, 1989; Berenji, 1991).

2.2.1 The KBANN Algorithm

As a reminder, the goal of this thesis is to generate a learning algorithm that is able to effectively use the available data, computer time, and background knowledge to generate the best concept possible. My approach is to follow the first step of Figure 5 of translating the domain theory into an initial guess for an appropriate network topology (using the Kbann algorithm), then continually refine this topology to find the best network (or set of networks) for my concept. This section explains the Kbann algorithm.

Kbann works by translating a domain theory consisting of a set of propositional rules directly into a neural network (see Figure 6). Figure 6a shows a Prolog-like rule set that defines membership in category a. Figure 6b represents the hierarchical structure of these rules, with solid lines representing necessary dependencies and dotted lines representing prohibitory dependencies. Figure 6c represents the network Kbann creates from this

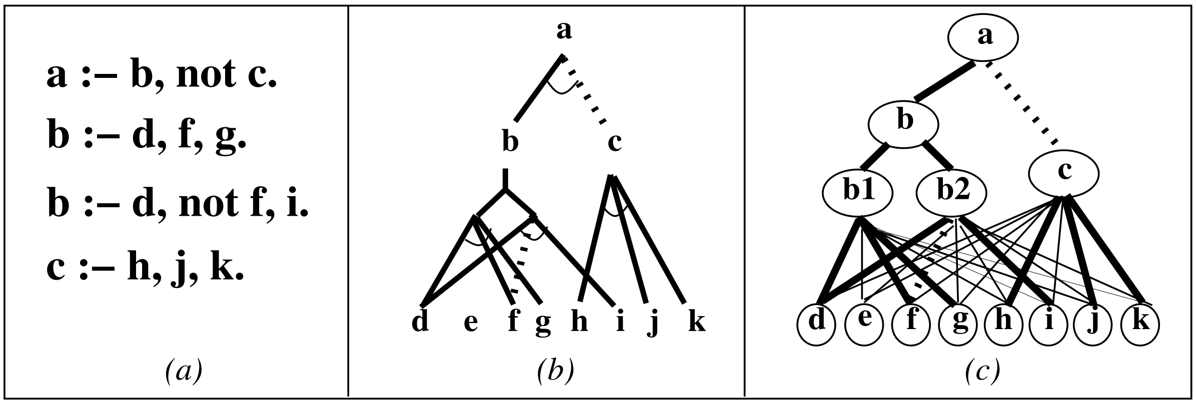


Figure 6: KBANN's translation of a knowledge base into a neural network. Panel (a) shows a sample propositional rule set in Prolog (Clocksin & Mellish, 1987) notation, panel (b) illustrates this rule set's corresponding dependency tree, and panel (c) shows the resulting network created by KBANN's translation.

translation. It sets the biases so that nodes representing disjuncts have an output near 1 only when at least one of their high-weighted antecedents is satisfied, while nodes representing conjuncts must have all of their high-weighted antecedents satisfied (i.e., near 1 for positive links and near 0 for negative links). Otherwise activations are near 0. KBANN creates nodes $b1$ and $b2$ in Figure 6c to handle the two rules deriving b in the rule set. The thin lines in Figure 6c represent low-weighted links that KBANN adds to allow these rules to add new antecedents during backpropagation training. Following network initialization, KBANN uses the available training instances to refine the network links. Refer to Towell (1991) or Towell and Shavlik (1994) for more details.

KBANN has been successfully applied to many real-world problems, such as the control of a chemical plant (Scott et al., 1992), protein folding (Maclin & Shavlik, 1993), finding genes in a sequence of DNA (Opitz & Shavlik, 1993; Towell & Shavlik, 1994), and ECG patient monitoring (Watrous et al., 1995). In each case, KBANN was shown to produce improvements in generalization over standard neural networks for small numbers of training examples. In fact, Towell (1991) favorably compared KBANN with a wide variety of algorithms, including purely symbolic theory-refinement systems, on the promoter and splice-junction tasks that I use as testbeds in this thesis.

2.2.2 Limitations of KBANN

While training the KBANN-created network alters the antecedents of existing rules, it does not have the capability of inducing new rules. For instance, KBANN is unable to add a new rule for inferring b in Figure 6's example. To help illustrate this point, consider the following example. Assume that the correct version of Figure 6a's domain theory also includes the rule:

$$b \text{ :- not } d,\ e,\ g.$$

Although I trained the KBANN network shown in Figure 6c with all possible examples of the correct concept, it was unable to completely learn the conditions when a is true.

Towell (1991) showed, with real-world empirical evidence (the DNA promoter domain), that because KBANN is unable to induce new rules, its generalization performance suffers when given "impoverished" domain theories – theories that are missing rules or antecedents needed to adequately learn the true concept. While real-world domains are clearly useful when exploring the utility of an algorithm, they are difficult to use in closely controlled studies that examine different aspects of an algorithm. An artificial domain, however, allows one to know the relationship between the theory provided to the learning system and the correct domain theory. Thus I repeat Towell's experiments using an artificial domain, so that I can more confidently determine how much KBANN suffers when given impoverished domains.

My artificial domain is derived from the game of chess. The concept to be learned is those board configurations where moving a king one space forward is illegal (i.e., the king would be in check). To make the domain tractable, I only consider a 4x5 subset of the chess board (shown in Figure 7). The king is currently in position $c1$ and the player is considering moving it to position $c2$ (which is currently empty) and thus wants to know if this is a legal move. Pieces considered in this domain are a queen, a rook, a bishop,

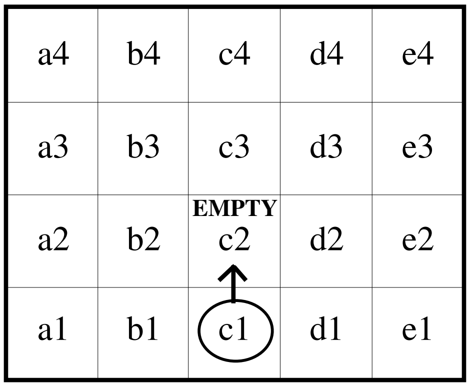


Figure 7: The 4x5 subset of a chess board used to investigate a weakness of KBANN. The king is currently on position $c1$, and will be moved to position $c2$ if this is a legal move. This move is illegal if the king would be in check on position $c2$.

---

and a knight for both sides. (Appendix A contains a detailed description of the rule set defining this domain.)

In order to investigate the types of corrections at which KBANN is effective, I ran experiments where I perturbed the correct domain theory in various ways, then gave these incorrect domain theories to KBANN. I perturbed the domain theory in four different ways: (a) adding an antecedent to a rule, (b) deleting an antecedent from a rule, (c) adding a rule, and (d) deleting a rule. This perturbation was done by scanning each antecedent (rule), and probabilistically deciding whether to add another antecedent (rule), delete it, or leave it alone.

I created a new rule by first setting its consequent to be the consequent of the scanned rule. I then added from two to four randomly chosen antecedents, since the number of antecedents of each rule in the correct rule base also ranged from two to four. I added antecedents to both newly created and existing rules by randomly selecting from the consequents below the rule[2] as well as the input features. When deleting rules, I considered only rules whose consequent was not that of the final conclusion. Finally, after I deleted antecedents, if a rule's consequent does not appear in another rule's antecedent

[2]A consequent is "below" a rule if it has a longer path to the target node (i.e., the final conclusion) in the rule base's dependency tree (e.g., see Figure 6b).

list (and is not the final conclusion), then this rule was deleted as well. I did this because it is unlikely that a domain expert would provide undefined intermediate conclusions in a domain theory.

Figure 8 shows the test-set error (i.e., error measured on a set of examples not seen by the learning algorithm) that results from perturbing the chess domain in each of these four ways. This error is the average of five runs of five-fold cross validation (Stone, 1974; Breiman et al., 1984) Each five-fold cross validation is run by first splitting the data into five equal-sized sets. Then, five times one set is held out while the remaining four are used to train the system. The held-out set is used to measure test-set performance of KBANN's final concept, thus giving an estimate of its generalization performance. I ran multiple (five) cross-validation experiments in order to dampen fluctuations due to the random partitioning of the data, and to judge the statistical significance of the results.

Figures 8a and 8b demonstrate that KBANN is effective at correcting spurious additions to the domain theory; however, Figures 8c and 8d show that KBANN's generalization degrades rapidly when rules and antecedents are deleted from the domain theory. KBANN's high error rate in Figure 8d is partially due to the fact that when I deleted antecedents, I also deleted rules whose consequent no longer appeared in another rule's antecedent list (unless it was the final conclusion, `illegal-move`). As a point of comparison, my theory-corruption algorithm deleted 26% of the rules when it removed 50% of the antecedents.

These artificial-domain results strengthen Towell's (1991) findings on real-world domains that while KBANN is effective at removing extraneous rules and antecedents, its generalization ability suffers when rules or antecedents are missing. The reason for this behavior results from the fact that KBANN is unable to modify its topology; thus, if many rules are missing, KBANN has to compress all required rules into few hidden nodes, perhaps beyond its capacity. An ideal algorithm, therefore, must be able to dynamically expand the topology of its KNN during training. This is the focus of the next chapter.

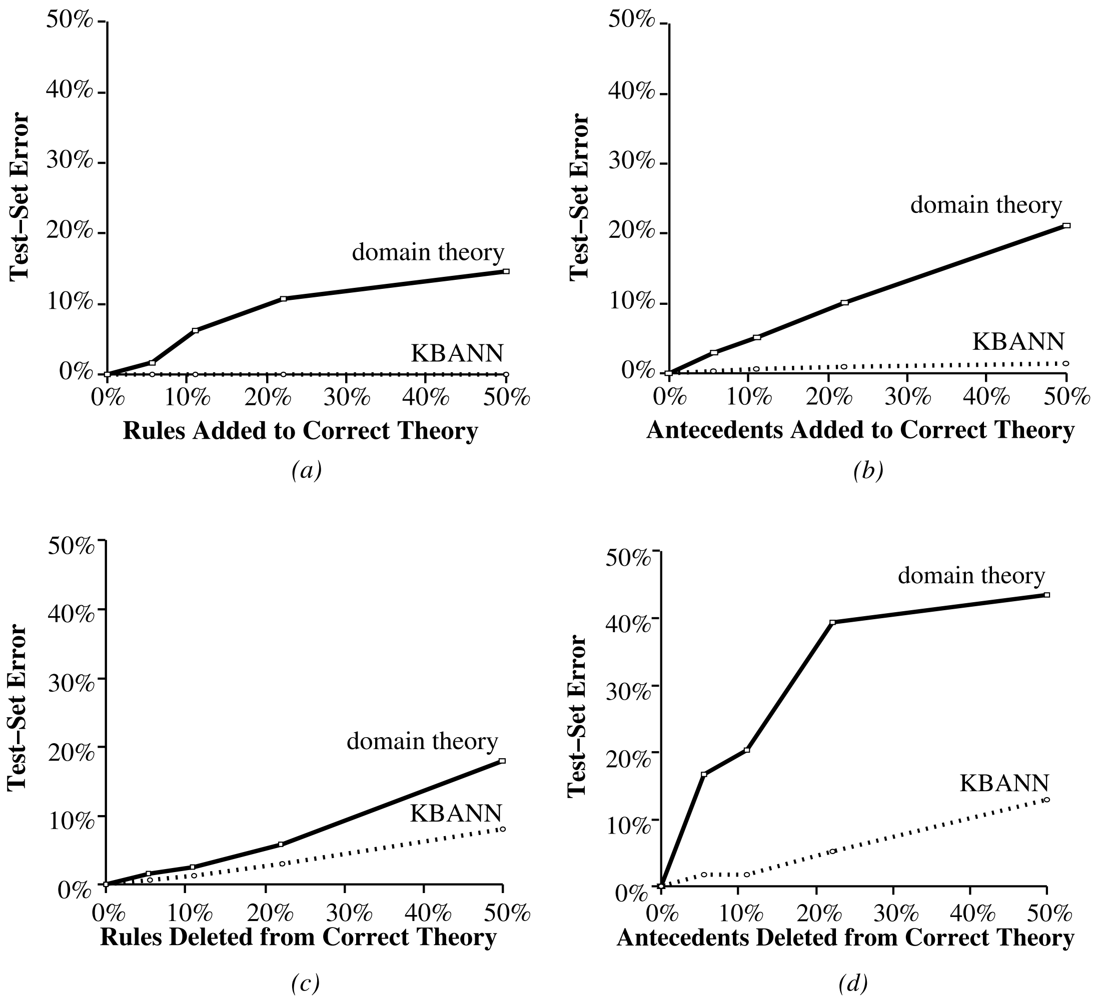


Figure 8: Impact on generalization of perturbing four different ways the correct chess-like domain theory. In all graphs, the top line (labeled *domain theory*), is the test-set error of the domain theory given to KBANN, while the bottom line (labeled KBANN) is the test-set error of KBANN after backpropagation training. The test-set errors were obtained from five runs of five-fold cross validation.

Chapter 3

Expanding Knowledge-Based Neural Networks

I indicated in Chapter 2 that connectionist theory-refinement systems produce *knowledge-based* neural networks that have been shown to frequently generalize better than many other inductive-learning and theory-refinement systems. I also demonstrated, however, that if these systems do not dynamically alter the topology of their networks during training, then they are restricted in the types of refinements they can make to the domain theory. Thus with sparse domain theories, generalization suffers, and the original rules must be significantly altered in order to account for the training data. While it is clearly important to classify the examples as accurately as possible, changes to the initial domain theory should be kept to a minimum because the domain theory presumably contains useful information, even if it is not completely correct. *Hence, the goal in this chapter is to expand, during the training phase, knowledge-based neural networks so that they are able to learn the training examples without needlessly corrupting their initial rules.*

My first algorithm, TopGen (Topology Generator), heuristically searches through the space of possible expansions of a knowledge-based network, guided by the symbolic domain theory, the network, and the training data. It does this expansion by adding hidden nodes to the neural representation of the domain theory. More specifically, it first uses a symbolic interpretation of the trained network to help locate the primary errors of the network, and then adds nodes to this network in a manner analogous to adding rules and conjuncts to a propositional rule base. I show that adding hidden nodes in this fashion synergistically combines the strengths of refining the rules symbolically with the strengths of refining them with backpropagation. Thus, TopGen mainly differs from other network-growing algorithms (Fahlman & Lebiere, 1989; Mezard & Nadal, 1989; Frean, 1990) in that it is specifically designed for knowledge-based neural networks.

TopGen tries to effectively use available computer time by employing a beam search, rather than a faster hill-climbing algorithm, to better find a good network topology. Finding such a topology allows increased generalization, provides the network with the ability to learn without overly corrupting the initial set of rules, and increases the interpretability of the network so that efficient rules may be extracted. Section 3.3 presents evidence for these claims.

3.1 The TopGen Algorithm

As stated above, the focus of TopGen, is to be able to introduce new rules into a knowledge-based neural network. It introduces new rules by heuristically searching through the space of possible ways of adding nodes to the network, trying to find the network that best refines the initial domain theory (as measured using "validation sets"[1]). Briefly, TopGen looks for nodes in the network with high error rates, and then adds new nodes to these parts of the network.

Table 2 summarizes the beam-search-based TopGen algorithm. TopGen uses two validation sets, one to evaluate the different network topologies, and one to help decide where new nodes should be added (it also uses the second validation set to decide when to stop training individual networks). TopGen uses KBANN's rule-to-network translation

[1] TopGen splits the training set into a set of training instances and two sets of validation instances. Note that the instances in these three sets are separate from a *test* set that experimenters commonly set aside to test the generalization performance of a learning algorithm.

Table 2: The TopGen algorithm.

TopGen:

GOAL: Find the network best describing the domain theory and training set.

1. Disjointly separate the training set into a training set and two validation sets (`validation-set-1` and `validation-set-2`).
2. Train, using backpropagation, the initial network produced by KBANN's rules-to-network translation and put on the *OPEN* list.
3. Until *stopping criterion* reached:
   (a) Remove the best network, according to `validation-set-2`, from the *OPEN* list.
   (b) Use **ScoreEachNode** to determine the *N* best places to expand the topology.
   (c) Create *N* new networks, train and put on the *OPEN* list.
   (d) Prune the *OPEN* list to length *M*.
   (e) Report best network seen so far according to `validation-set-2`.

ScoreEachNode:

GOAL: Suggest good ways to add new nodes using error in `validation-set-1`.

1. Temporarily use the threshold ("step") activation function at each node.
2. Score each node in the network as follows:
   (a) Set each node's `correctable-false-negative` and `correctable-false-positive` counters to 0.
   (b) For each misclassified example in `validation-set-1`, cycle through each node and determine if modifying the output of that node will correctly classify the example, incrementing the counters when appropriate.
3. Use the counters to order possible node corrections. High `correctable-false-negative` counts suggest adding a disjunct, while high `correctable-false-positive` counts suggest adding a conjunct.

algorithm to create an initial guess for the network's topology. This network is trained using backpropagation (Rumelhart et al., 1986) and is placed on an OPEN list. In each cycle, TopGen takes the best network (as measured by `validation-set-2`) from the OPEN list, decides possible ways to add new nodes, trains these new networks, and places them on the OPEN list. This process is repeated until reaching either (a) a `validation-set-2` accuracy of 100% or (b) a previously set time limit. TopGen reports the best new network, according to `validation-set-2`, whenever one is found.

3.1.1 *Where* Nodes Are Added

TopGen must first find nodes in the network with high error rates.[2] It does this by scoring each node (which corresponds to a rule in a symbolic domain theory) using examples from `validation-set-1`. By using examples from this validation set, TopGen adds nodes on the basis of where the network fails to generalize, not where it fails to memorize the training set.

TopGen makes the empirically verified assumption that almost all of the nodes in a trained knowledge-based network are either fully active or inactive. Towell (1991), as well as self-inspection of my networks, has shown this to be a valid assumption. The reason for this is that the network's weights are large and, hence, the nodes are usually operating in the "flat" portions of their activation function. By making this assumption, each non-input node in a TopGen network can be treated as a step function (or a Boolean rule) so that errors have an all-or-nothing aspect, thus concentrating topology refinement on misclassified examples, not on erroneous portions of *every* example.

TopGen keeps two counters for each node, one for false negatives and one for false positives, defined with respect to each individual node's output, not the final output. An example is considered a false negative if it is incorrectly classified as a negative example, while a false positive is one incorrectly classified as a positive example. TopGen

[2]It is important to note that methods for finding nodes with high error rates are just heuristics to help guide the search of where to add new nodes; TopGen is able to backtrack if a "bad" choice is made since it uses beam search.

increments counters by recording how often changing the "Boolean" value of a node's output leads to a misclassified example being properly classified. That is, if a node is active for an erroneous example and changing its output to be inactive results in correct classification for the example, then the node's false-positives counter is incremented. TopGen increments a node's false-negatives counter in a similar fashion. By checking for single points of failure, TopGen looks for rules that are *near misses* (Winston, 1975).

After the counter values have been determined, TopGen sorts these counters in descending order, breaking ties by preferring nodes farthest from the output node. TopGen then creates N new networks, where each network contains a single correction (as determined by the first N sorted counters). The larger the value for N, the more "breadth" TopGen has in its search. (I set N to two in this thesis.) Section 3.1.2 details *how* the nodes are added, based on the counter type (i.e., a false-negative or false-positive counter) and node type (i.e., an AND or OR node).

I also tried other approaches for blaming nodes for error, but they did not work as well on my testbeds. One such method is to propagate errors back by starting at the final conclusion and recursively considering an antecedent of a rule to be incorrect if both (a) its consequent is incorrect and (b) the antecedent does not match its "target." I approximate targets by starting with the output node, then recursively considering a node to have the same target as its parent, if the weight connecting them is positive, or the opposite target, if this weight is negative. While this method works for symbolic rules, TopGen suffers under this method because its antecedents are weighted. Antecedents with small-weighted links are counted as much as antecedents with large-weighted links. Because of this, I also tried using the backpropagated error to blame nodes, however backpropagated error becomes too diffuse in networks having many layers, such as the ones often created by TopGen.

3.1.2 *How* Nodes Are Added

Once TopGen estimates where it should add new nodes, it needs to know *how* to add these nodes. TopGen makes the assumption that when training one of its networks, the meaning of a node does not shift significantly. Making this assumption allows us to alter the network in a fashion similar to refining symbolic rules. Towell (1991) demonstrated that making a similar assumption about KBANN networks was valid.

Figure 9 illustrates the possible ways TopGen adds nodes to one of its networks. In a symbolic rule base that uses negation-by-failure (Rich, 1983, Chapter 5). one can decrease false negatives by either dropping antecedents from existing rules or adding new rules to the rule base. I showed (in Section 2.2.2) that KBANN is effective at removing antecedents from existing rules, but is unable to add new rules. Therefore, TopGen adds nodes, intended for decreasing false negatives, in a fashion that is analogous to adding a new rule to the rule base. If the existing node is an OR node, TopGen adds a new node as its child (see Figure 9a), and fully connects this new node to the input nodes. When the existing node is an AND node, TopGen creates a new OR node that is the parent of the original AND node and another new node that TopGen fully connects to the inputs (see Figure 9c); TopGen moves the outgoing links of the original node (A in Figure 9c) to become the outgoing links of the new OR node.

In a symbolic rule base, one can decrease false positives by either adding antecedents to existing rules or removing rules from the rule base. In Section 2.2.2, I demonstrated that KBANN can effectively remove rules, but it is less effective at adding antecedents to rules and is unable to invent (constructively induce, Michalski, 1983) new terms as antecedents. Thus TopGen adds new nodes, intended to decrease false positives, in a fashion that is analogous to adding new constructively induced antecedents to the network. Figures 9b and 9d illustrates how this is done (analogous to Figures 9a and 9c explained above). These mechanisms allow TopGen to add to the domain theory rules whose consequents were previously undefined.

TopGen handles nodes that are neither AND nor OR nodes by deciding if such a node

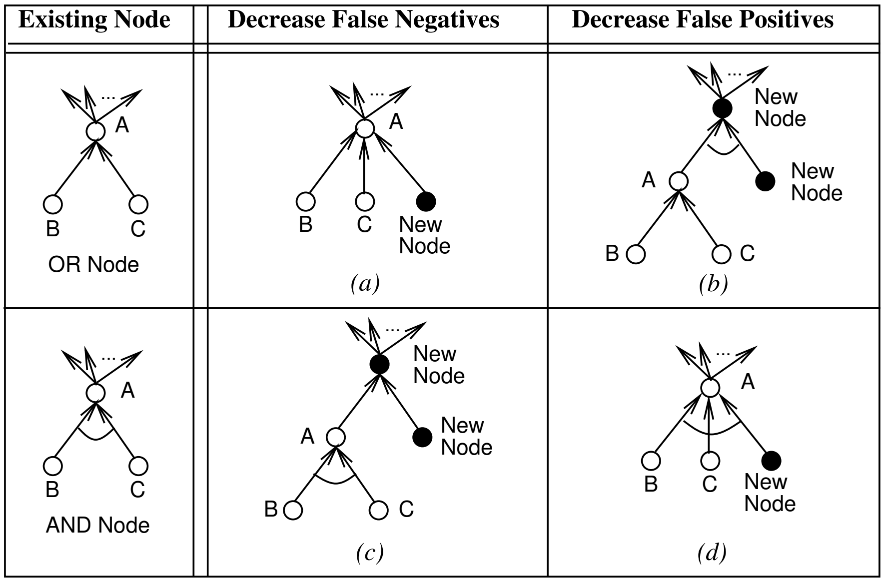


Figure 9: Possible ways to add new nodes to a knowledge-based neural network (arcs indicate AND nodes). To decrease false negatives, we wish to broaden the applicability of the node. Conversely, to decrease false positives, we wish to further constrain the node.

---

is "closer" to an AND node or an OR node. An AND node has an activation near 1 only when *all* of its high-weighted antecedents are correctly activated (i.e., near 1 for positive links and near 0 for negative links). When this happens the node's weighted input will be approximately the sum of the positive links; otherwise, one of the antecedents will be incorrectly activated and the input will be less than this. Therefore, an AND node's bias must be slightly less than the sum of its *positive* incoming weights. Alternately, an OR node has an activation near 1 as long as *at least one* of its high-weighted antecedents is correctly activated. When all the antecedents are incorrectly activated, the node's input is the sum of the high-weighted negative links; otherwise, at least one of the antecedents is correctly activated and the input will be greater than this. Therefore, an OR node's bias must be slightly greater than the sum of its *negative* incoming weights. Thus, if a node's bias is closer to the summed positive weights than to the summed negative weights, I consider it an AND node; otherwise, I consider it an OR node.

3.1.3 Additional Algorithmic Details

After new nodes are added, TopGen must retrain the network. While I want the new weights to account for most of the error, I also want the old weights to change if necessary. That is, I want the older weights to retain what they have previously learned, while at the same time move in accordance with the change in error caused by adding the new node. In order to address this issue, TopGen multiplies the learning rates of existing weights by a constant amount (I use 0.5) every time new nodes are added, producing an exponential decay of learning rates. (In backpropagation, each inter-node link can have its own learning rate.)

I also do not want to change the domain theory unless there is considerable evidence that it is incorrect. That is, there is a trade-off between changing the domain theory and disregarding the misclassified training examples as noise. To help address this, TopGen uses a variant of weight decay (Hinton, 1986). Weights that are part of the original domain theory decay toward their initial value, while other weights decay toward zero.

My weight-decay term, then, decays weights as a function of their distance from their initial values and is a slight variant of the term proposed by Weigend et al. (1990) - see Equation 12 in Section 2.1.1 for more details. I express this distribution as follows:

$$P(N) = exp\left[-\sum_{ij}\frac{(w_{ij}-w_{init_{ij}})^2}{1+(w_{ij}-w_{init_{ij}})^2}\right].$$

(I set σ_1 and σ_2 in Equation 12 to 1 in this thesis.)

3.2 Example of TopGen

To help illustrate TopGen's method for adding rules, I return to the example given in Section 2.2.2. Assume we are given the domain theory in Figure 10a (this is a repeat of Figure 6 in Section 2.2.1). I indicated in Section 2.2.2 that if the correct "target" theory we are trying to learn should have also included the rule:

$$b \text{ :- not } d, e, g.$$

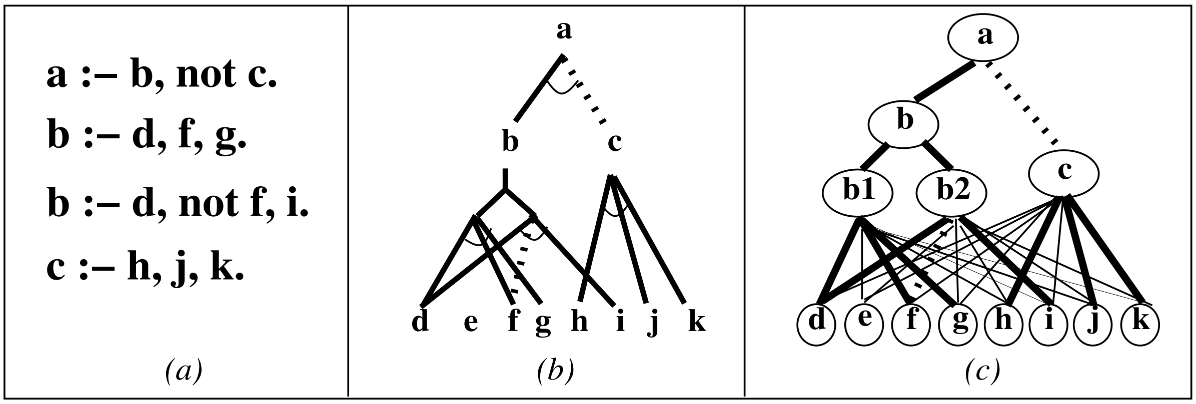


Figure 10: KBANN's translation of a knowledge base into a neural network. Panel (a) shows a sample propositional rule domain theory in Prolog (Clocksin & Mellish, 1987) notation, panel (b) illustrates this rule set's corresponding dependency tree, and panel (c) shows the resulting network created from KBANN's translation. (This is a repeat of Figure 6 in Section 2.2.1.)

---

then KBANN is unable to completely learn when a is true. TopGen, however, is able to learn this concept; it begins by training the KBANN network, obtaining little improvement to the original rule base. It then proceeds by using misclassified examples from `validation-set-1` to find places where adding nodes could be beneficial, as explained below.

Consider the following positive example of category a, which is incorrectly classified by the domain theory,

$$\neg\, d \wedge e \wedge f \wedge g \wedge \neg\, h \wedge \neg\, i \wedge j \wedge k.$$

While node c (from Figure 10c) is correctly false in this example, node b is incorrectly false. B is false since both $b1$ and $b2$ are false. If b had been true, this example would have been correctly classified (since c is correct), so TopGen increments the `correctable-false-negative` counter for b. TopGen also increments the counters of $b1$, $b2$, and a, using similar arguments.

Nodes a, b, $b1$, and $b2$ will all have high `correctable-false-negative` counters after all the examples are processed. Given these high counts, TopGen considers adding OR nodes to nodes a, $b1$, and $b2$, as done in Figure 9c, and also considers adding another disjunct, analogous to Figure 9a, to node b. Any one of these four corrections allows the network to learn the target concept. Since TopGen breaks ties by preferring nodes farthest from the output node, it prefers $b1$ or $b2$.

3.3 Empirical Results

In this section I test TopGen on five domains: the artificial chess-related domain introduced in Section 2.2.2, and four real-world problems from the Human Genome Project. (Refer to Appendix A for more information on these domains.)

3.3.1 Chess Results

In Section 2.2.2, I showed that KBANN's generalization suffered when it was given a domain theory where rules or antecedents were deleted from the correct theory. In this section, I test TopGen's performance on these two cases to see how well it addresses these limitations of KBANN.

To help test the efficiency of TopGen's approach for choosing where to add hidden nodes, I also compare its performance with a simple approach (referred to as *Strawman* hereafter) that adds one layer of fully connected hidden nodes "off to the side" of the KBANN network. (Strawman is similar to Fletcher and Obradovic's algorithm, which I further discuss in Section 6.1.1.) Figure 11 illustrates the topology of such a network. The topology of the original KBANN network remains intact, while I add extra hidden nodes in a fully connected fashion between the input and output nodes. If a domain theory is impoverished, it is reasonable to hypothesize that simply adding nodes in this fashion would increase performance. Strawman trains 50 different networks (using weight decay), ranging from 0 to 49 extra hidden nodes and, like TopGen, uses a validation set to choose the best network.

My initial experiment addresses the generalization ability of TopGen when given

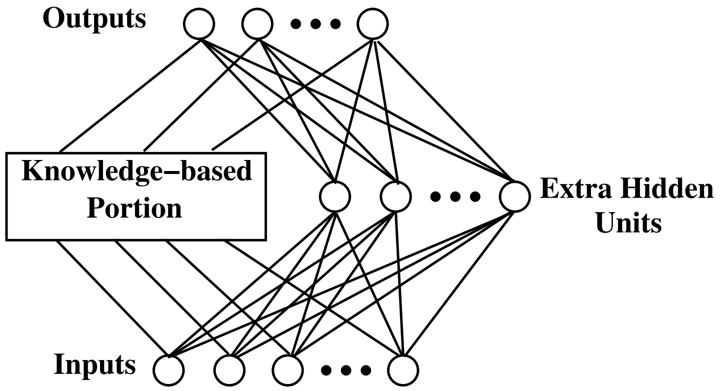


Figure 11: Topology of the networks used by Strawman.

domain theories that are missing rules or antecedents. Figure 12 reports the test-set error when I randomly delete rules and antecedents from the chess domain theory (see Section 2.2.2 for details on how I generated these domain theories). The plotted results are the averages of five runs of five-fold cross-validation.

The top horizontal line in each graph results from a fully connected, single-layer, feed-forward neural network. For each fold, I trained various networks containing up to 100 hidden nodes and used a validation set to choose the best network. This line is horizontal because the networks do not use any of the domain theories. Graphing this line gives a point of reference for the generalization performance of a standard neural network; thus, comparisons to knowledge-based neural networks show the utility of being able to exploit the domain theory.

The next two curves in each graph report KBANN's and Strawman's performance; notice that Strawman produced almost no improvement over KBANN in either case. Finally, TopGen, the bottom curve in each graph, had a significant increase in accuracy, having an error rate of about half that of both KBANN and Strawman. Like Strawman, TopGen considered 50 networks during its search. As a point of comparison, when 50% of the rules were deleted, TopGen added 22.4 nodes on average, while Strawman added 22.2 nodes; when 50% of the antecedents were deleted, TopGen added 21.2 nodes, while Strawman added 5.1 nodes.

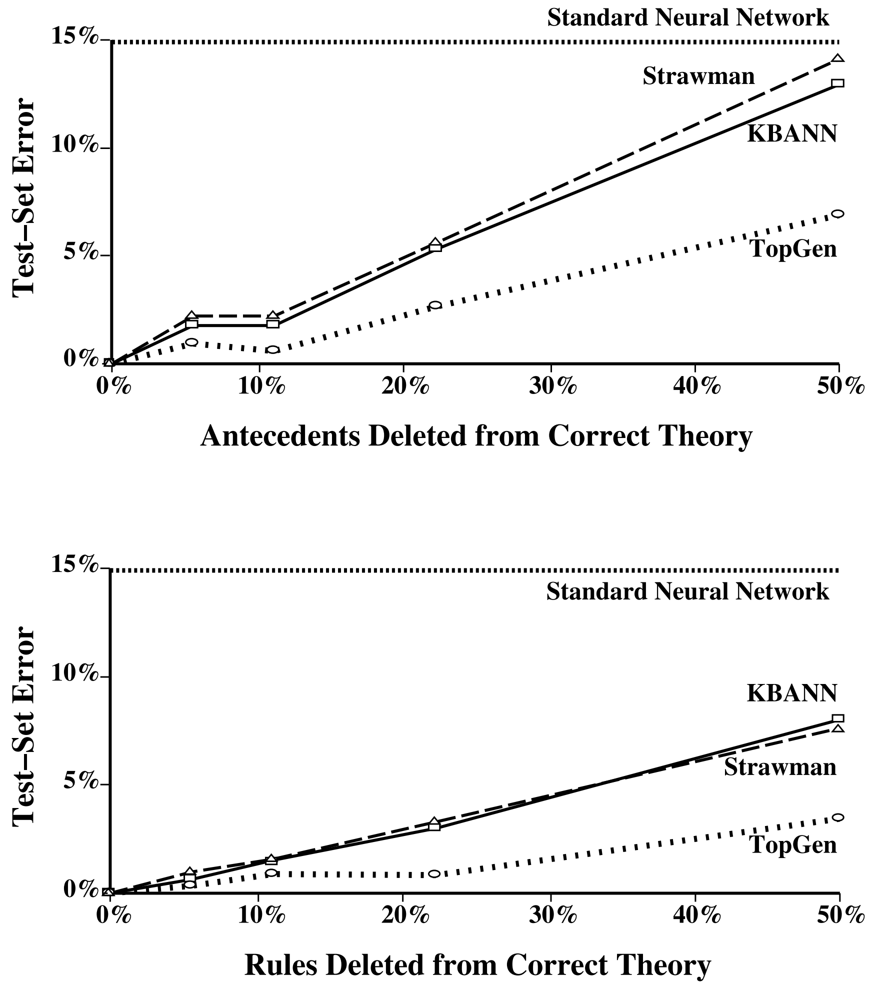


Figure 12: Test-set error of four learners on the chess problem. In both graphs, the Y-axis is the mean test-set error obtained from five runs of five-fold cross validation. One-tailed, pairwise t-tests indicate that the difference between TopGen and KBANN and the difference between TopGen and Strawman are significant at the 95% confidence level at all nonzero percentages, except when 6% of the rules were deleted.

Domain-Theory Corruption

As stated earlier, it is important to correctly classify the examples while deviating from the initial domain theory as little as possible. Since the domain theory may have been inductively generated from past experiences that are not present in our current set of training instances, one should deviate from the meaning of this theory as little as possible. Therefore, I measure semantic distance, rather than syntactic distance, when deciding how far a learning algorithm has deviated from the initial domain theory. Also, syntactic distance is difficult to measure, especially if the learning algorithm generates rules in a different form than the initial domain theory. However, one can estimate semantic distance by considering only those examples in the test set that the original domain theory classifies correctly. Error on these examples indicates how much the learning algorithm has corrupted *correct* portions of the domain theory.

Figure 13 shows accuracy on the portion of the test set where the original domain theory is correct. When the initial domain theory has few missing rules or antecedents (less than 15%), neither TopGen, KBANN, nor Strawman overly corrupt this domain theory in order to compensate for these missing rules; however, as more rules or antecedents are deleted, both KBANN and Strawman corrupt their domain theory much more than TopGen does.

3.3.2 DNA Results

I also ran TopGen on four problems from the Human Genome Project. Each of these problems aid in locating genes in DNA sequences, and are described in more detail in Appendix A. Briefly, the first domain, *promoter recognition*, contains 234 positive examples, 4,921 negative examples, and 31 rules. The second domain, *splice-junction determination*, contains 3,190 examples distributed among three classes, and 23 rules. The third domain, *ribosome binding sites* (RBS), contains 366 positive examples, 1,511 negative examples, and 17 rules. Finally, the last domain, *transcription termination sites*, contains 142 positive examples, 5,178 negative examples, and 61 rules.

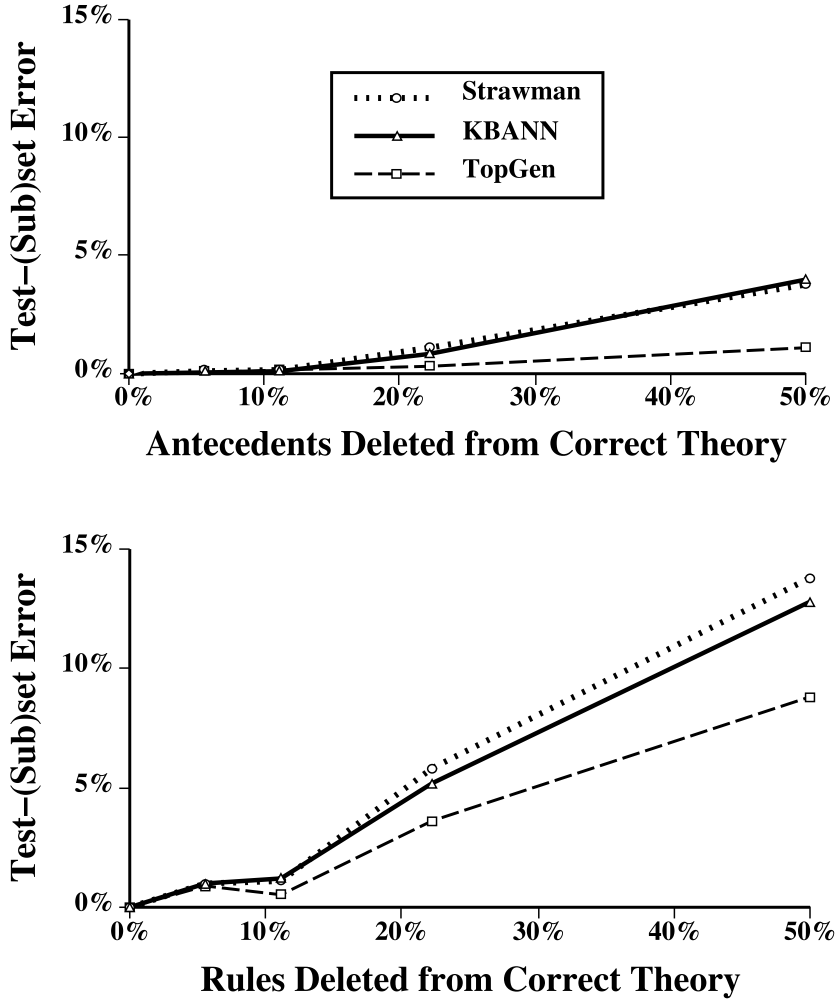


Figure 13: Error rate of the various learners on the subset of the test set where the corrupted domain theory is correct. In both graphs, the Y-axis is the mean test-set error obtained from five runs of five-fold cross validation. Pairwise, one-tailed t-tests indicate that TopGen differs from both KBANN and Strawman at the 95% confidence level when more than 12.5% of the rules or antecedents are deleted.

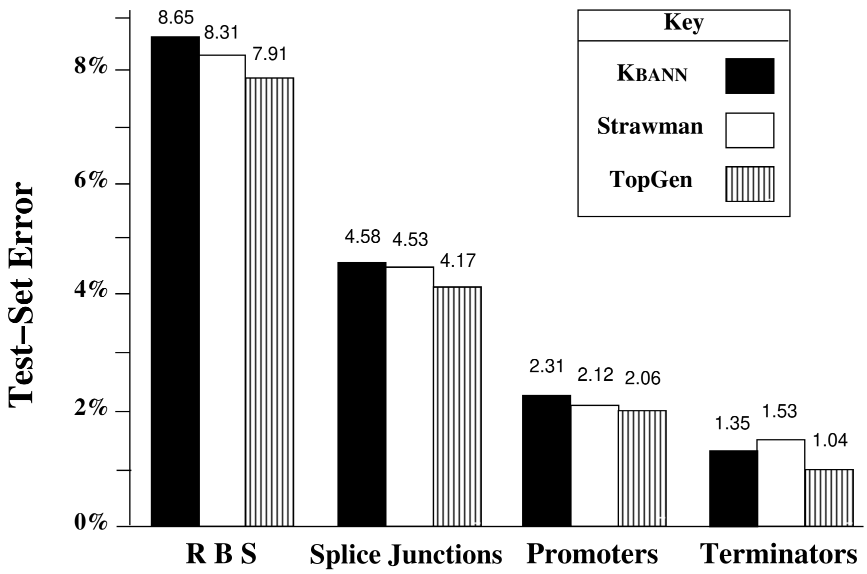


Figure 14: Error rates on four Human Genome problems.

My first experiment addresses the test-set accuracy of TopGen on these domains. The results in Figure 14 demonstrate that TopGen generalizes better than does both KBANN and Strawman. These results are averages of five runs of five-fold cross-validation, where TopGen and Strawman consider 30 networks during their beam search. Pairwise, one-tailed t-tests indicate TopGen differs from both KBANN and Strawman at the 95% confidence level on all four domains, except with Strawman on the promoter domain.

Table 3 shows that TopGen and Strawman added about the same number of nodes on all domains, except the terminator dataset. On this dataset, adding nodes off to the side of the KBANN network, in the style of Strawman, usually decreases accuracy. Therefore, whenever Strawman picked a network other than the KBANN network, its generalization usually decreased. Even with Strawman's difficulty on this domain, TopGen was still able to effectively add nodes to increase performance.

Table 3: Total number of nodes added (on average).

Domain	TopGen	Strawman
RBS	8.2	8.0
Splice Junction	4.0	5.2
Promoters	4.4	5.0
Terminators	9.4	1.2

Considering a Large Number of Networks

Given the lengthy training times of each network in the above results, TopGen only considers 30 networks during its search. In order to trace the behavior of TopGen as it considers even more networks, I decrease the training time of each network by using only a subset of the possible examples, then increase TopGen's search to 500 networks. (Given the size of KBANN's terminator network, I was still unable to use this domain.) The reduced splice-junction domain contains 1,200 examples distributed equally among three classes. I then randomly selected a subset of the negative examples for the other two domains so that they contained a 3-to-1 ratio of negative-to-positive examples. Thus the reduced promoter domain contains 702 negative examples, while the reduced RBS domain contains 1,098 negative examples.

Given the smaller datasets, keeping two validation sets from the training data can severely hinder the generalization ability of each network since few instances remain to train these networks. Therefore, I ran each algorithm without using `validation-set-1`. Instead, I avoided overfitting by only training each network for a few (e.g., about ten) epochs, and estimated the error of each node based on the training set. (To check the validity of this approach, I also ran TopGen using both validation sets, but as expected, generalization suffered; however, if enough data is available, one should keep both sets.) Note that it is harder for TopGen to reduce KBANN's error on smaller datasets, since a larger percentage of the training set must be set aside to ensure each network can be accurately scored.

Figure 15 gives the test-set error of TopGen as it searches through the space of network topologies. The results are from a ten-fold cross validation. The horizontal line in each graph results from the KBANN algorithm. Even though KBANN considers only one network, I drew a horizontal line for the sake of visual comparison. While TopGen is still able to improve KBANN on these reduced datasets, its improvement is primarily limited to the first 50 networks considered.

3.4 Discussion of TopGen

Towell (1991) has demonstrated that KBANN generalizes better than many machine learning algorithms on the promoter and splice-junction domains, including purely symbolic approaches to theory refinement. Yet, even though a domain expert (M. Noordewier) believed the four Human Genome domain theories were large enough for KBANN to adequately learn the concepts, TopGen is able to effectively add new nodes to the corresponding network. The effectiveness of adding nodes in a manner similar to reducing error in a symbolic rule base is verified with comparisons to a naive approach for adding nodes. If a KBANN network, resulting from an impoverished domain theory, suffered only in terms of capacity, then adding nodes between the input and output nodes would have been just as effective as TopGen's approach to adding nodes. The difference between TopGen and this naive approach (Strawman) is particularly pronounced on the terminator dataset.

TopGen has a longer runtime than KBANN; however, TopGen tries to continually improve the quality of its solution over time, and can produce its currently best answer whenever one is needed. TopGen's runtime is dominated by the training process of each network; thus, TopGen's runtime is longer than KBANN's by approximately the number of networks it considers during its search. If a quick answer is needed, TopGen just degenerates to the KBANN algorithm.

With the reduced DNA datasets, I showed that while TopGen initially decreases generalization error, improvements are primarily limited to the first few (i.e., 50) networks

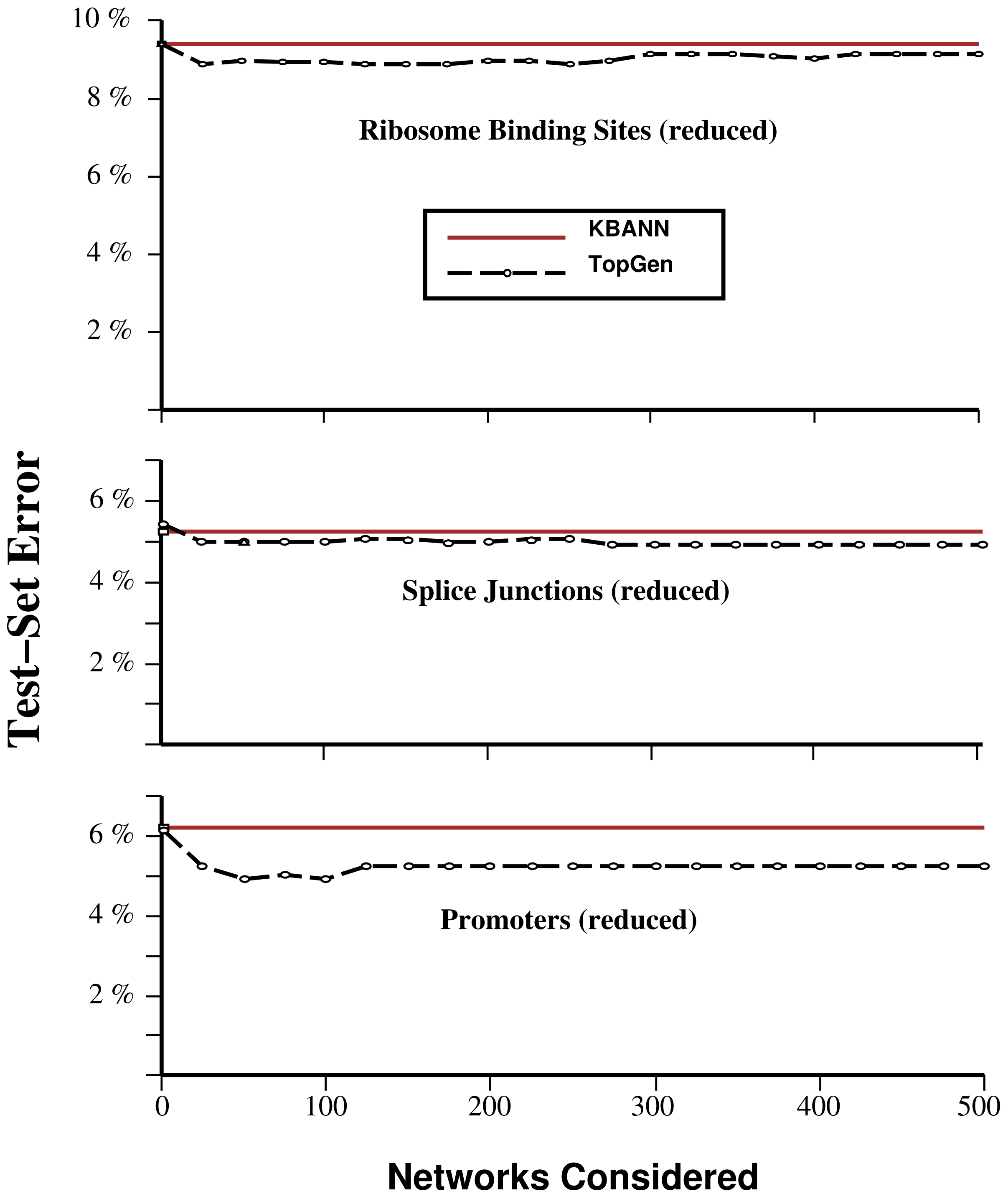


Figure 15: Error rates on the three reduced Human Genome problems.

considered in the search. Therefore, TopGen is not so much an "anytime" algorithm, but rather is a first step towards one. This is mostly due to the fact that TopGen only considers larger networks that contain the original KBANN network as subgraphs; however, as one increases the number of networks considered, one should also increase the variety of networks considered during its search. In fact, broadening the range of networks considered during the search is the major focus of the next chapter. It is important to note, though, that TopGen is effective in its initial refinements, and should therefore be considered as an initial method for any system that attempts to broaden TopGen's search.

3.5 Future Work on TopGen

While I present future work within the general framework of my thesis in Chapter 7, I give future plans specific to TopGen here. One such direction is testing new ways of adding nodes to the network. Newly added nodes are currently fully connected to all the input nodes. Other possible approaches include: adding them to only a portion of the inputs; adding them to nodes that have a high correlation with the error; adding them to the next "layer" of nodes.[3]

Another area of future work includes extensively testing other approaches for locating error. Even though this is only a heuristic to help guide the search, a good heuristic will allow more efficient search of the hypothesis space. Methods of using the back-propagated error as well as symbolic techniques for determining error have been tested, but did not improve performance, for reasons explained in Section 3.1.1. A future research direction includes trying variants of these techniques.

As indicated in Section 2.2, it is desirable to understand what network has learned. Therefore, future work is to use a rule-extraction algorithm (Fu, 1991; Towell & Shavlik,

[3]Although one can define *layer* many different ways, I define a node's layer to be the longest path from it to an output node.

1993; Craven & Shavlik, 1994b) to measure the interpretability of a refined TopGen network. I hypothesize that TopGen builds networks that are more interpretable than naive approaches of adding nodes, such as the approach taken by Strawman. Trained KBANN networks are interpretable because (a) the meaning of its nodes does not significantly shift during training and (b) almost all the nodes are either fully active or inactive (Towell & Shavlik, 1993). Not only does TopGen add nodes in a symbolic fashion, it adds them in a fashion that does not violate these two assumptions. I discuss this in more detail in Section 7.2.4.

3.6 Wrap-up

TopGen, my new algorithm that I presented in this chapter, heuristically searches through the space of possible expansions of the original network, guided by the symbolic domain theory, the network, and the training data. It does this by adding hidden nodes to the neural representation of the domain theory, in a manner analogous to adding rules and conjuncts to the symbolic rule base. Experiments indicate that TopGen is able to heuristically find effective places to add nodes to the knowledge bases of four real-world problems, as well as an artificial chess domain. Therefore, TopGen is successful in overcoming KBANN's limitation of not being able to dynamically add new nodes. In doing so, my system increases KBANN's ability to generalize and learn a concept without needlessly corrupting the initial rules. (Versions of this chapter were previously published in Opitz and Shavlik, 1993, 1995a, 1995b.)

Chapter 4

Genetically Refining Knowledge-Based Neural Networks

I demonstrated in Section 2.2.2 that the generalization performance of a knowledge-based neural network (KNN) suffers hen created from a domain theory that is missing rules needed to adequately learn the true concept. Last chapter, I directly addressed this problem with TopGen, a beam-search-based method that heuristically searches for good places to add rules to the neural representation of the provided domain theory. I showed that while TopGen is initially effective at adding nodes, its increase in generalization primarily occurs in the first few networks considered. This behavior largely results from the fact that TopGen only generates networks that are *expansions* of the original KNN; thus, if one has time to consider a large number of networks, one should also increase the variety of networks they consider. In this chapter, I present a second algorithm, Regent (REfining, with Genetic Evolution, Network Topologies), that addresses this limitation by using genetic algorithms (GAs) to broaden the type of topologies that TopGen considers during its search. In doing so, Regent is able to consider both larger and smaller refinements of the original Kbann network.

GAs are a logical choice for broadening a search since they have been shown to be efficient in their use of global information (Holland, 1975; Goldberg, 1989; Koza, 1992).

REGENT proceeds by first trying to generate, from the domain theory, a diversified initial population. It then produces new candidate networks via variants of the genetic operators of *crossover* and *mutation* that were specifically designed for KNNs. For instance, REGENT's crossover operator tries to maintain the rule structure of the crossed-over networks, while its mutation operator uses TopGen's method for adding new nodes to a KNN. Experiments reported in this chapter show that REGENT is able to better search for network topologies than is TopGen if it has time to consider many candidate networks.

Before I present the REGENT algorithm, however, I give a short overview of GAs. I first explain their motivation and give a high-level overview of how they work. I then briefly explain the predominant theory of *why* GAs work as well as they do. Those already familiar with GAs should skip to Section 4.2.

4.1 Introduction to Genetic Algorithms

Genetic algorithms are based on the theory of Darwinian evolution (Darwin, 1859). Briefly, the evolutionary process occurs when there is a population of self-reproducing individuals that differ in their ability to survive. Each individual has chromosomes that make up its structure and behavior. Fitter individuals (i.e., individuals that are best able to perform tasks needed in the environment) survive longer and reproduce at a higher rate. Over a period of time and many generations, the population as a whole contains more individuals whose chromosomes are descended from fit relatives.

4.1.1 How Genetic Algorithms Work

GAs as a method of machine learning began in the 1960's with John Holland (1975). The key idea behind GAs centers around selective breeding of a set of individuals that comprise a population. This population is usually kept at a fixed size N and, as I will soon address, it is important to maintain "diversity" within the population to ensure a broad search of the space of possible individuals. In the spirit of "survival of the

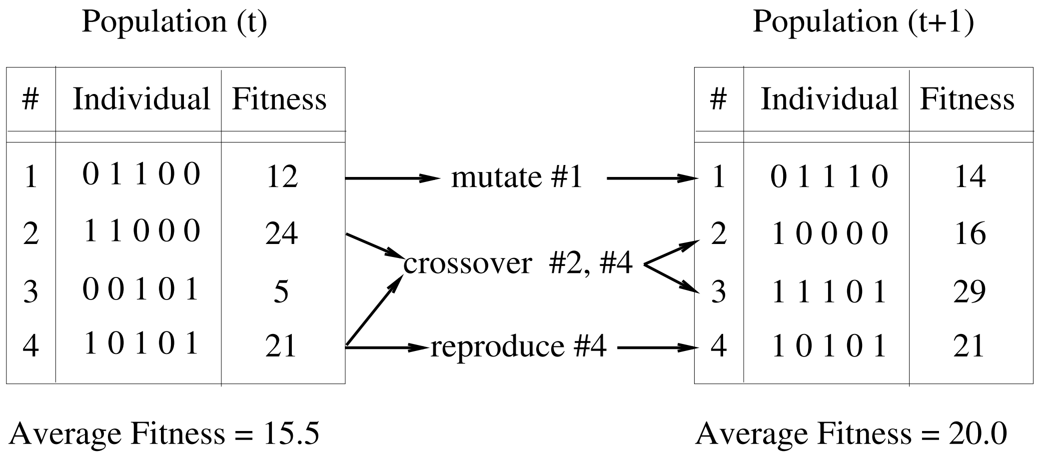


Figure 16: An example of one generation in a standard GA. A population of four individuals is reproduced, through fitness-proportional reproduction, to create another population of four individuals.

---

fittest," individuals are picked proportional to their fitness to create offspring which will be inserted into a new population (possibly replacing members of the current population).

Figure 16 presents an example of one generation of this process. The size of the population in this case is four, and the genetic operators of crossover, mutation, and reproduction (explained below) produce the next generation of individuals. In "classical" GAs, the individuals in the population are represented as fixed-length strings of binary digits (e.g., Figure 16 has a bit-string whose length is five). This bit string is the analogy of chromosomes in living organisms and is the blueprint for the construction of that individual. While designing a good bit-string representation for a problem allows one to use existing genetic operators, some problems, such as finding real-valued weights for a neural network, may not be conducive for such a representation. In these cases, it is often better to design problem-specific genetic operators for a different representation.

While the *genotype* of an individual is the bit-string that is manipulated during the genetic process, the *phenotype* is the personification of the individual. This phenotype is problem dependent, and once it has been defined, one must define a method for testing it to determine its fitness. The fitness of each individual in Figure 16 is simply defined

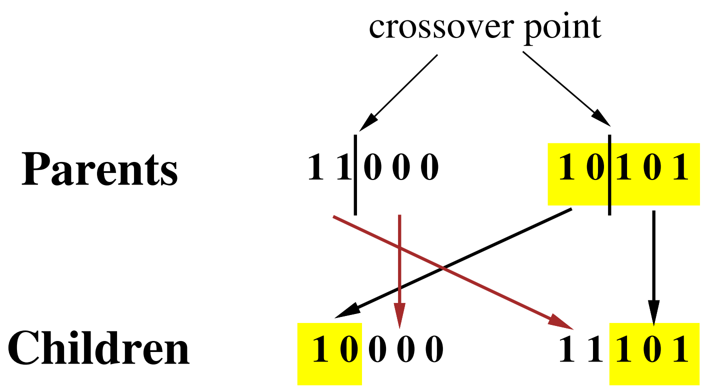


Figure 17: An example of the "classical" crossover operator.

to be the binary encoding of that individual. Therefore the fittest possible individual in this case is comprised of all ones and has a fitness of 31.

An individual's fitness determines the expected number of offspring it will conceive. The classical way of proportionally picking individuals to breed is through "roulette wheel" selection. An individual is allocated a slot on the wheel that has a width proportional to its fitness. Parents are selected from the population by spinning the wheel. Thus highly fit individuals are more likely to be chosen during each spin.

The most common operators for creating a new generation of individuals are *reproduction*, *crossover*, and *mutation*. Reproduction works by simply copying an individual to the next generation, and is used in the cases like Figure 16 where each generation is a distinctly different population. An alternate approach (which I use in this thesis) is to keep one fixed-sized population. This is done by continually creating new members with the crossover and mutation operators, and replacing existing members of the population with the new individuals. Many replacement strategies, such a simply replacing the lowest-fit individual, have been proposed (Goldberg, 1989).

Figure 17 illustrates the typical crossover operator for fixed-length bit strings (these are the individuals crossed over in Figure 16). Crossover works by first picking two parents at random proportional to their fitness. A cut-point is then randomly picked for each parent and two children are created, each obtaining one segment from each parent.

Mutation is an operator designed to add new genetic traits to a population. For instance, it is possible that no members of a population contain a desirable bit value in a particular location. Since the crossover operator only *combines* two member of a population, it has no way of introducing new bits. For instance, there is no way to crossover two members of Figure 16 to generate an individual that has a one in the third bit. Mutation is typically a secondary operation that is only sparingly used (Goldberg, 1989).

4.1.2 Why Genetic Algorithms Work

The power of genetic search lies in its continual recombination of good *building blocks.* The underlying assumption is that fit individuals are made from good parts. If a particular combination of attributes is repeatedly found in highly fit individuals, we begin to think that this combination is at least partly responsible for this performance. The same is true for combinations repeatedly found in low-fit individuals. Genetic algorithms, then, are the embodiment of this intuitive approach for finding the attribute combinations responsible for good behavior in complex non-linear systems.

Holland (1975) mathematically showed the power of GAs with his Schema Theorem. A schemata is a string consisting of the original alphabet, augmented by the "don't care" symbol (*). For instance, two strings contain the schema "1 1 * 1 0" – "1 1 0 1 0" and "1 1 1 1 0." Different schemata have varying impact on the fitness of the strings which contain them. In our example, the strings containing the schema "1 * * * *" would always be more fit than the strings containing the schema "0 * * * *." We therefore expect "1 * * * *" to become more prominent in our population over time.

Briefly, the Schema Theorem states that, due to fitness-proportional reproduction, GAs perform near optimal exploration of all schemata partitions simultaneously. For instance, with second-order schemata (i.e., two bits are defined), there are four partitions. When deciding from which of these partitions to choose the next string in our search, there is a trade-off between continued exploration and greedy exploitation. We want to emphasize the partition where the average fitness of previously considered strings is highest, while at the same time continuing exploration in the other partitions. Fitness-proportional reproduction optimally allocates future individuals from all schema partitions simultaneously, based on the mean and variance of past fitnesses. Operators such as crossover and mutation do provide minor disruptions to this optimality, however (Goldberg, 1989).

Initially, it was thought (Liepins & Vose, 1990; Wilson, 1991) that GAs would perform well on any problem where there was little *deception* – problems where low-order schemata give misleading information about the probable average fitness of higher-order refinements. For instance, this could happen if the partition containing the optimal individual also contained many unfit individuals. Later, however, Forrest and Mitchell (1993) showed that GAs perform poorly on complex problems where the basic building blocks consist of many (four or more) bits, or where the basic blocks get split during crossover. Seeding the initial population with a domain theory (as Regent does) can help define the basic building blocks for these problems. Holland (1975), Goldberg (1989), and Koza (1992) provide further reading on GAs.

4.2 The REGENT Algorithm

My new algorithm, Regent, tries to broaden the types of networks that TopGen considers with the use of GAs. I view Regent as having two phases: (a) genetically searching through topology space, and (b) training each network using backpropagation. Regent uses the domain theory to aid in both phases. It uses the theory to help guide its search through topology space and to give a good starting point in weight space.

Table 4 summarizes the Regent algorithm. Regent first sets aside a *validation* set (from part of the *training* instances) for use in scoring the different networks. It then perturbs the Kbann-produced network to create an initial set of candidate networks. Next, Regent trains these networks using backpropagation and places them into the

Table 4: The REGENT algorithm.

GOAL: Search for the best network topology describing the domain theory and data.

1. Set aside a validation set from the training instances.
2. Perturb the KBANN-produced network in multiple ways to create initial networks, then train these networks and place them into the population.
3. Loop forever:
   (a) Create new networks using the crossover or mutation operator.
   (b) Train these networks with backpropagation, score with the validation set, and place into the population.
   (c) If a new network is the network with the lowest validation-set error seen so far (breaking ties by preferring the smallest network), report it as the current best concept.

---

population. In each cycle, REGENT creates new networks by crossing over and mutating networks from the current population that are randomly picked proportional to their fitness (i.e., validation-set correctness). It then trains these new networks and places them into the population. As it searches, REGENT keeps the network that has the lowest validation-set error as the best concept seen so far, breaking ties by choosing the smaller network in an application of Occam's Razor. A parallel version trains many candidate networks at the same time using the Condor system (Litzkow et al., 1988), which runs jobs on idle workstations.

A diverse initial population will broaden the types of networks REGENT considers during its search; however, since the domain theory may provide useful information that may not be present in the training set, it is still desirable to use this theory when generating the initial population. REGENT creates diversity about the domain theory by randomly perturbing the KBANN network at various nodes. REGENT perturbs a node by either deleting it, or by adding new nodes to it in a manner analogous to one of TopGen's four methods for adding nodes. (If there are multiple theories about a domain, all of them can be used to seed the population.)

4.2.1 REGENT's Crossover Operator

REGENT crosses over two networks by first dividing the nodes in each parent network into two sets, A and B, then combining the nodes in each set to form two new networks (i.e., the nodes in the two A sets form one network, while the nodes in the two B sets form another). Table 5 summarizes REGENT's method for crossover and Figure 18 illustrates it with an example. REGENT divides nodes, one level[1] at a time, starting at the level nearest the output nodes. When considering a level, if either set A or set B is empty, it cycles through each node in that level and randomly assigns it to either set. If neither set is empty, nodes are probabilistically placed into a set. The following equation calculates the probability of a given node being assigned to set A:

$$Prob(node\ i \in setA) = \frac{\sum_{j \in A} |w_{ji}|}{\sum_{j \in A} |w_{ji}| + \sum_{j \in B} |w_{ji}|}, \tag{14}$$

where $j \in A$ means node j is a member of set A and w_{ji} is the weight value from node i to node j. The probability of belonging to set B is one minus this probability. With these probabilities, REGENT tends to assign to the same set those nodes that are heavily-linked together. This helps to minimize the destruction of the rule structure of the crossed-over networks, since nodes belonging to the same syntactic rule are connected by heavily-linked weights. Thus, REGENT's crossover operator produces new networks by crossing-over rules, rather than simply crossing-over nodes.

REGENT must next decide how to connect the nodes of the newly created networks. First, a new network inherits all weight values from its parents that connect two nodes that either it inherited, or are input or output nodes. It then adds low-weighted links between unconnected nodes on consecutive levels. Finally, it adjusts the bias of all AND or OR nodes to help maintain their *original* function. For instance, if REGENT removes a positive incoming link for an AND node, it decrements the node's bias by subtracting

[1] Recall that a node's *level* is defined as the longest path from it to an output node.

Table 5: REGENT's method for crossing over networks.

Crossover Two Networks:

GOAL: Crossover two networks to generate two new network topologies.

1. Divide each network's hidden nodes into sets `A` and `B` using **DivideNodes**.
2. Set `A` forms one network, while set `B` forms another. Each new network is created as follows:
   (a) A network inherits weight w_{ji} from its parent if nodes i and j either are (i) also inherited or (ii) input or output nodes.
   (b) Link unconnected nodes between levels with near-zero weights.
   (c) Adjust node biases to keep original AND or OR function of each node (see text for explanation).

DivideNodes:

GOAL: Divide the hidden nodes into sets `A` and `B`, while probabilistically maintaining each network's rule structure.

While some hidden node is not assigned to set `A` or `B`:

(i) Collect those unassigned hidden nodes whose output is linked only to either previously-assigned nodes or output nodes.

(ii) **If** set `A` or set `B` is empty:
For each node collected in part (i), randomly assign it to set `A` or set `B`.
Else
Probabilistically add the nodes collected in part (i) to set `A` or set `B`.
Equation 14 shows the probability of being assigned to set `A`.
The probability of being assigned to set `B` is one minus this value.

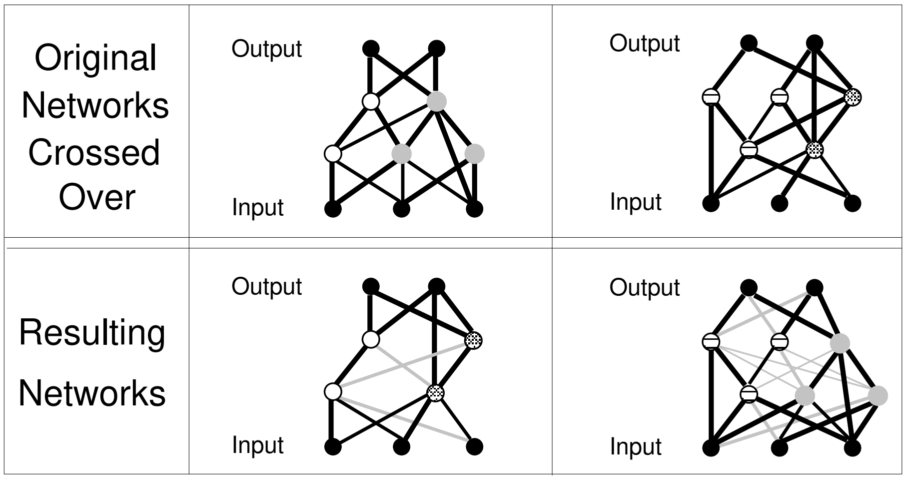


Figure 18: REGENT's method for crossing over two networks. The hidden nodes in each original network are divided into the sets `A` and `B`; the nodes in the two `A` sets form one new network, while the nodes in the two `B` sets form another. Grey lines represent low-weighted links that are added to fully-connect neighboring levels.

---

the product of the link's magnitude times the average activation (over the set of training examples) entering that link. REGENT increments the bias for an OR node by a similar amount when it removes negative incoming links.

4.2.2 REGENT's Mutation Operator

REGENT mutates networks by applying a variant of TopGen. REGENT uses TopGen's method for incrementing the false-negatives and false-positives counters for each node. REGENT then adds nodes, based on the values of these counters, the same way TopGen does. Since neural learning is effective at removing unwanted antecedents and rules from KNNs (see Section 2.2.2), REGENT only considers adding nodes, and not deleting them, during mutation. Thus, this mutation operator adds diversity to a population, while still maintaining a directed, heuristic-search technique for choosing where to add nodes; this directedness is important because I currently am unable to evaluate more than a few thousand possible networks per day.

4.2.3 Additional Details

REGENT adds newly trained networks to the population only if their validation-set correctness is better than or equal to an existing member of the population. When REGENT replaces a member, it chooses the member having the lowest correctness (ties are broken by choosing the oldest member). Other techniques (Goldberg, 1989), such as replacing the member nearest the new candidate network, can promote diverse populations; however, I do not want to promote diversity at the expense of decreased generalization. As a future research topic, I plan to investigate incorporating diversity-promoting techniques once I am able to consider tens of thousands of networks.

REGENT can be considered a Lamarckian[2], genetic-hillclimbing algorithm (Ackley, 1987), since it performs local optimizations on individuals, then passes the successful optimizations on to offspring. The ability of individuals to learn can smooth the fitness landscape and facilitate learning (Hinton & Nowlan, 1987). Thus, Lamarckian learning can lead to a large increase in learning speed and solution quality (Farmer & Belin, 1992; Ackley & Littman, 1994).

4.3 Experimental Results

In this section, I test REGENT on three real-world problems from the Human Genome Project. I first directly compare REGENT with TopGen. I then investigate adding randomly created networks to REGENT's initial population. Finally, I examine the utility of REGENT's genetic operators.

I ran REGENT on three of the Human Genome Project problems presented in Appendix A. In order to track the behavior of REGENT over many generations, I use the reduced datasets from Section 3.3.2 for *all* experiments in this chapter. As a reminder, the reduced promoter domain contains 234 positive examples, 702 negative examples,

[2]Lamarckian evolution is a theory based on the inheritance of characteristics acquired during a lifetime.

and 31 rules. The reduced splice-junction domain contains 1,200 examples distributed equally among three classes, and 23 rules. Finally, the reduced ribosome binding sites (RBS) domain, contains 366 positive examples, 1,098 negative examples, and 17 rules. All results in this chapter are from a ten-fold cross validation; in each fold, REGENT is run with a population size of 20.

4.3.1 Generalization Ability of REGENT

This section's experiments compare the test-set accuracy (i.e., generalization) of REGENT with TopGen's. Figure 19 shows the test-set error of KBANN, TopGen, and REGENT as they search through the space of network topologies. The horizontal line in each graph results from the KBANN algorithm. As was the case in the previous chapter, I drew a horizontal line for the sake of visual comparison. The first point of each graph, after one network is considered, is nearly the same for all three algorithms, since they all start with the KBANN network; however, TopGen and REGENT differ slightly from KBANN since they must set aside part of the training set to score their candidate networks. Notice that TopGen stops improving after considering 10 to 30 networks and that the generalization ability of REGENT is better than TopGen's after this point.

Figure 20 presents the test-set error of TopGen and REGENT after they each consider 500 candidate topologies. The standard neural network results are from a fully-connected, single-layer, feed-forward neural network; for each fold, I trained various networks containing up to 100 hidden nodes and used a validation set to choose the best network. My results show KBANN generalizes much better than the best of these standard networks, thus further confirming KBANN's effectiveness in generating good network topologies. While TopGen is able to improve on the KBANN network, REGENT is able to significantly decrease the error rate over both KBANN and TopGen.

Table 6 contains the number of hidden nodes in the final networks produced by KBANN, TopGen, and REGENT. The results demonstrate the REGENT produces networks that are larger than both KBANN's and TopGen's networks (even though TopGen

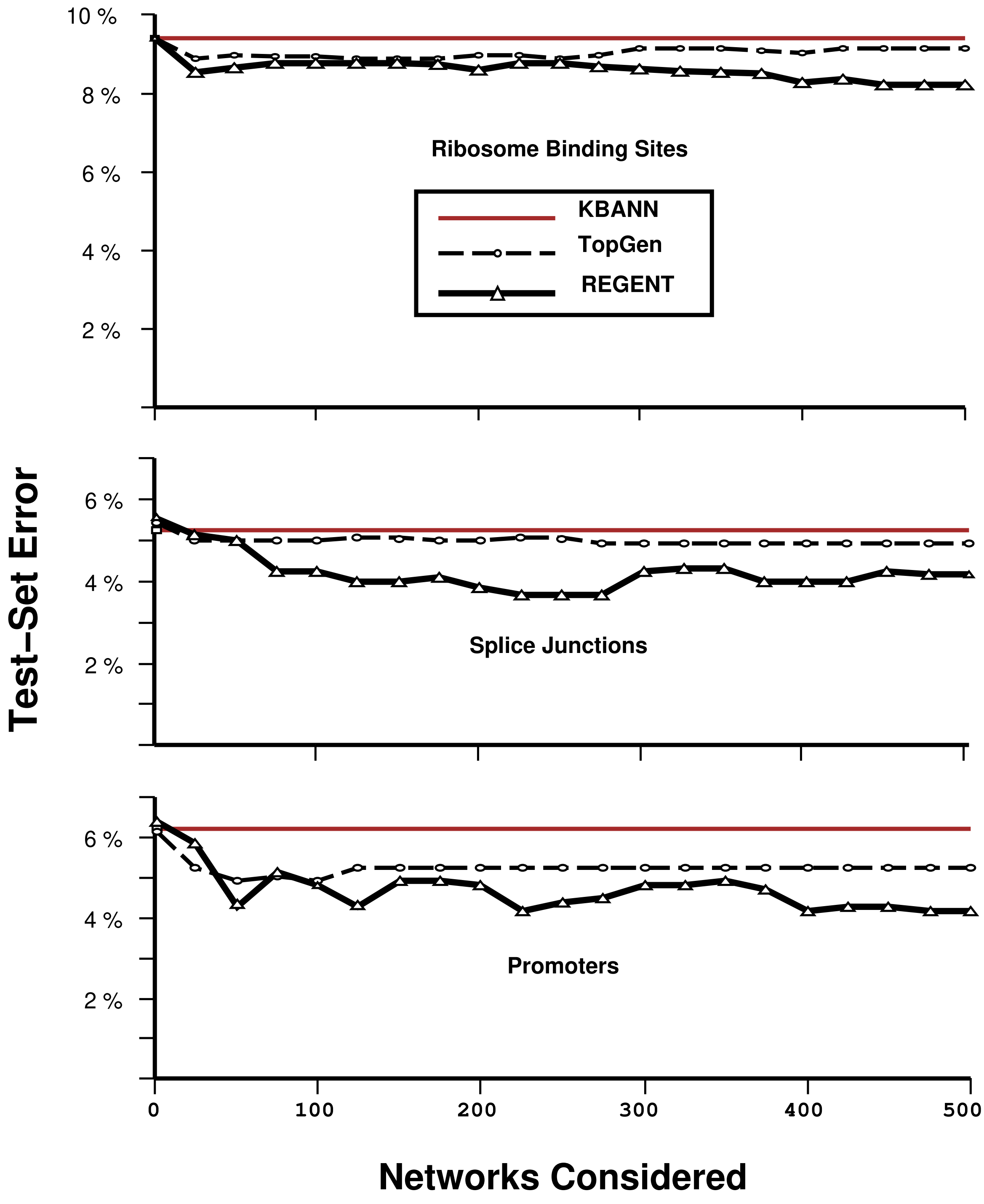


Figure 19: Error rates on the three reduced Human Genome problems.

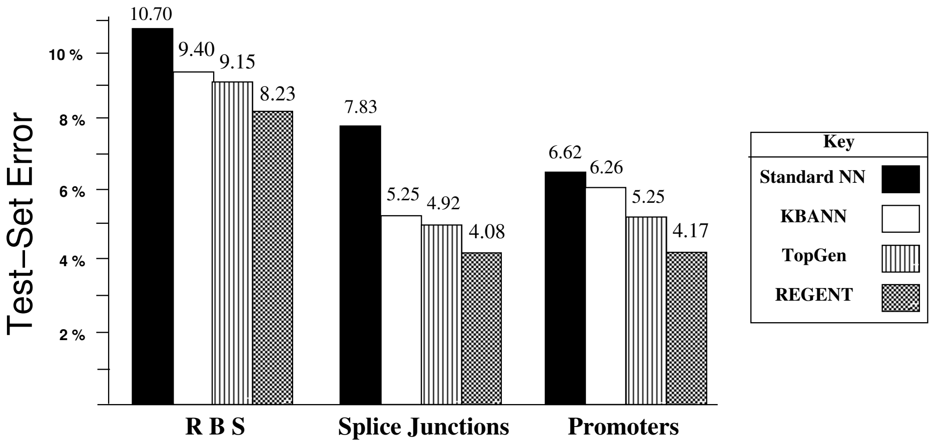


Figure 20: Error rates after TopGen and REGENT each consider 500 networks. Pairwise, one-tailed t-tests indicate that REGENT differs from both KBANN and TopGen at the 90% confidence level.

Table 6: Number of hidden nodes in the networks output by KBANN, TopGen, and REGENT. The columns show the mean number of hidden nodes found within these learning algorithm's networks. Standard deviations are contained within parentheses; I do not report the standard deviation for KBANN since there is only one network.

Domain	KBANN	TopGen	REGENT
RBS	18	42.1 (9.3)	70.1 (25.1)
Splice Junction	21	28.4 (4.1)	32.4 (12.2)
Promoters	31	40.2 (3.3)	74.9 (38.9)

only *adds* nodes during its search). While REGENT's networks are larger, it does not necessarily mean that they are more "complex." I inspected sample networks and found that there are large portions of the network that are either not used (e.g., they are insignificantly small) or are functional duplications of other groups of hidden nodes.

One could prune weights and nodes during REGENT's search; however, such pruning can prematurely reduce the variety of structures available for recombination during crossover (Koza, 1992). Real-life organisms, for instance, have superfluous DNA that are believed to enhance the rate of evolution (Watson et al., 1987; Suzuki et al., 1989). However, while pruning network size *during* genetic search may be unwise, one could prune Regent's final network using Hassibi and Stork's (1992) Optimal Brain Surgeon algorithm. This post-pruning process may increase the future classification speed of the network, as well as increase its comprehensibility (see Section 7.2.4 for more details).

4.3.2 Including Non-KNNs in REGENT's Population

The correct theory may be quite different from the initial domain theory. Thus, in this section I investigate whether one should include, in the initial population of networks, a variety of networks not obtained directly from the domain theory. Currently, Regent creates its initial population by always perturbing the Kbann network. To include networks that are not obtained from the domain theory, I first randomly pick the number of hidden nodes to include in a network, then randomly create *all* of the hidden nodes in this network. I do this by adding new nodes to a randomly picked output or hidden node using one of TopGen's four methods for adding new nodes (refer to Figure 9 in Section 3.1.2). Adding nodes in this manner creates random networks whose node structure is analogous to dependencies found in symbolic rule bases, thus creating networks suitable for Regent's crossover and mutation operators.

Table 7 shows the test-set error of Regent with various percentages of KNNs present in the initial population. The first row contains the results of running Regent with a purely random initial population (i.e., the population contains no KNNs). The second row lists the results when Regent creates half its population with the domain theory, and the other half randomly. Finally, the last row contains the results of seeding the entire population with the domain theory.

The results demonstrate that including, in the initial population, networks that were not created from the domain theory increases Regent's test-set error on all three domains. This occurs because the randomly generated networks are not as correct as the KNNs, and thus offspring of the original KNN quickly replace the random networks. Hence, diversity in the population suffers compared to methods that start with a whole population of KNNs. Assuming the domain theory is not "malicious," it is therefore

Table 7: Test-set error after considering 500 networks. Pairwise, one-tailed t-tests indicate that running REGENT with 100% KNNs differs from 0% KNNs at the 95% confidence level on all three domains; however, given the inherent similarity and lengthy run-times of these algorithms, the difference between the runs of 50% and 100% KNNs is not significant at this level.

	RBS	Splice Junction	Promoters
0% KNN	9.7%	6.3%	5.1%
50% KNN	8.6%	4.3%	4.6%
100% KNN	8.2%	4.1%	4.2%

better to seed the entire population from the KBANN network. Should the domain theory indeed be malicious and contain information that promotes spurious correlations in the data, it would then be reasonable to randomly create the "whole" population. Thus running REGENT both with and without the domain theory allows one to investigate the utility of that theory.

4.3.3 Value of REGENT's Mutation

Typically with GAs, mutation is a secondary operation that is only sparingly used; however, REGENT's mutation is a directed approach that heuristically adds nodes to KNNs in a provenly effective manner (i.e., it uses TopGen). It is therefore reasonable to hypothesize that one should apply the mutation operator more frequently than traditionally done in GAs. The results in this section test this hypothesis.

Figure 21 presents the test-set error of REGENT with varying percentages of mutation (versus crossover) when creating new networks in step 3a of Table 4. Each graph plots four curves: (a) 0% mutation (i.e., REGENT only uses crossover), (b) 10% mutation, (c) 50% mutation, and (d) 100% mutation. Performing no mutation tests the value of using only crossover, while 100% mutation tests the efficacy of the mutation operator by itself. Note that 100% mutation is just TopGen with a different search strategy; instead of keeping an OPEN list, a population of KNNs are generated and members of

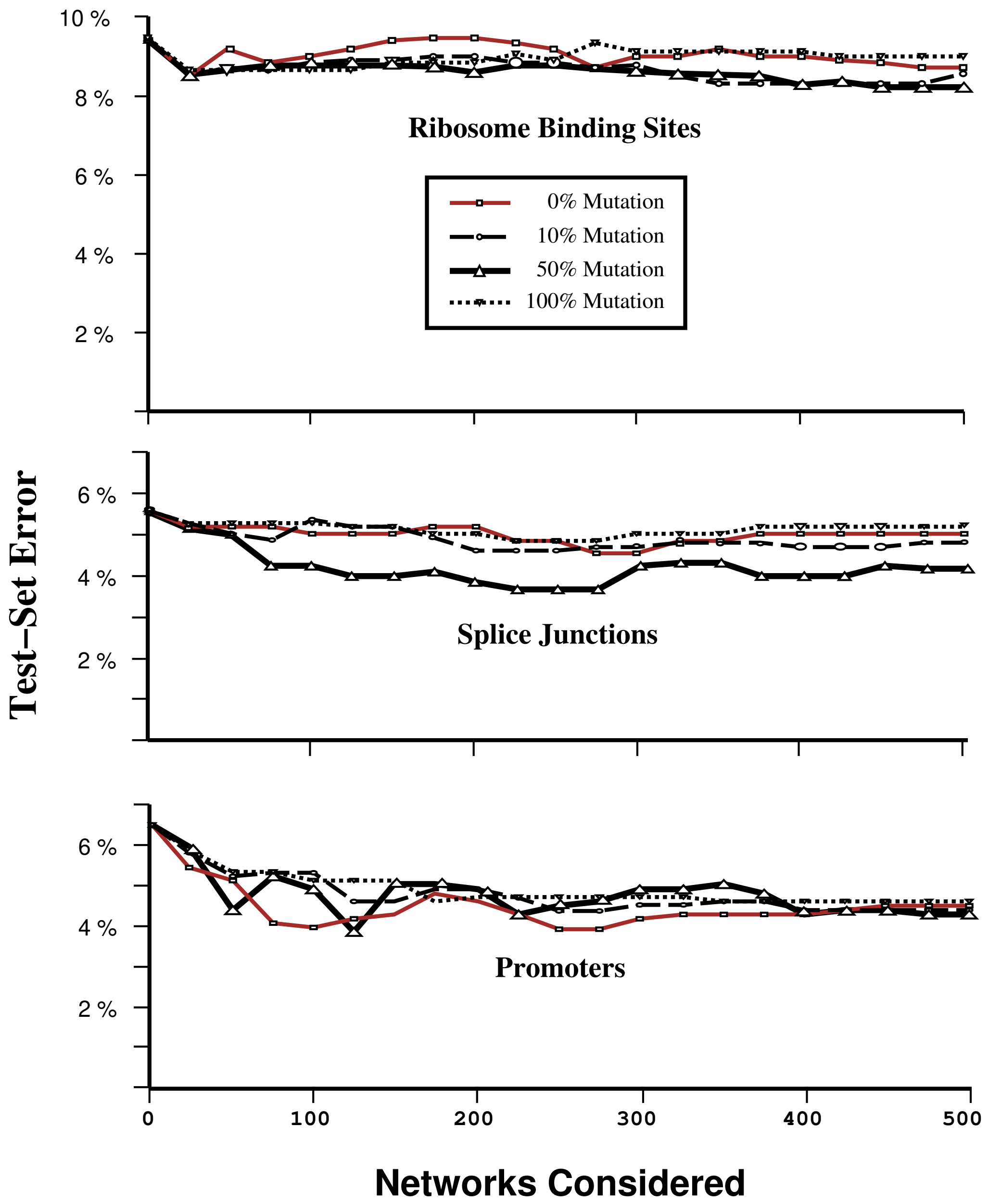


Figure 21: Error rates of REGENT with different fractions of mutation verses crossover after considering 500 networks. Given the inherent similarity of the algorithms, and the limited number of runs due to computational complexity, the results are not significant at the 95% confidence level.

the population are improved proportional to their fitness. The other two curves (10% and 50% mutation) test the synergy between the two operators. Performing 10% mutation is closer to the traditional GA viewpoint that mutation is a secondary operation, while 50% mutation means that both operations are equally valuable. (Previous experiments in this chapter used 50% mutation and 50% crossover.)

The results illustrate the synergy that exists between the two operations. Except for the middle portion of the promoter domain, the results show that, qualitatively, using both operations at the same time is better than using either operation alone. In fact, equally mixing the mutation and crossover operator is better than the other three curves on all three domains once REGENT has considered 500 networks. This result is particularly pronounced on the splice-junction domain.

4.3.4 Value of REGENT's Crossover

REGENT tries to cross over the rules in the networks, rather than just blindly crossing over nodes. It does this by probabilistically dividing the nodes in the network into two sets where nodes belonging to the same rule tend to belong to the same set. In this section, I test the efficacy of REGENT's crossover by comparing it to a variant of itself where it *randomly* assigns nodes to two sets (rather than using **DivideNodes** in Table 5).

Table 8 contains the results of this test after 250 networks were considered. In the first row, `REGENT-random-crossover`, REGENT randomly breaks its hidden nodes into two sets, while in the second row, REGENT assigns nodes to two sets according to Table 5. In both cases, REGENT creates half its networks with its mutation, and the other half with crossover. Although the differences are not statistically significant, the results suggest that keeping the rule structure of the networks intact during crossover is important; otherwise, the basic building blocks of the networks (i.e., the rules) get split during crossover, and studies have shown the importance of keeping intact the basic building blocks during crossover (Goldberg, 1989; Forrest & Mitchell, 1993).

Table 8: Test-set error of two runs of REGENT: (a) randomly crossing over "nodes" in the networks, and (b) one with crossing over "rules" in the network (defined by Equation 14). Both runs considered 250 networks and used half crossover, half mutation. The results are not significant at the 95% confidence level since there is only a slight difference between the learning algorithms and long run-times limited runs to a 10-fold cross validation.

	Promoters	Splice Junction	RBS
REGENT-random-crossover	4.6%	4.7%	9.1%
REGENT	4.4%	4.1%	8.8%

4.4 Discussion of REGENT

Previously I stated that KBANN has been shown to generalize better than many other machine learning algorithms on the promoter and splice-junction domains (the RBS dataset did not exist then). Despite this success, REGENT is able to effectively use available computer cycles to significantly improve generalization over both KBANN and last chapter's improvement to KBANN, the TopGen algorithm. REGENT reduces KBANN's test-set error by 12% for the RBS domain, 22% for the splice-junction domain, and 33% for the promoter domain; it reduces TopGen's test-set error by 10% for the RBS domain, 17% for the splice-junction domain, and 21% for the promoter domain. Also, REGENT's ability to use available computing time is further aided by its being inherently parallel, since I can train many networks simultaneously.

Further results show that REGENT's two genetic operators complement each other. The crossover operator considers a large variety of network topologies by probabilistically combining rules contained within two "successful" KNNs. Mutation, on the other hand, makes smaller, directed improvements to members of the population, while at the same time adding diversity to the population by adding new rules to the population. Equal use of both operators allow a wide variety of topologies to be considered, while at the same time being able to make incremental improvements to members of the population.

Since REGENT considers many networks, it can select a subset of the final population of networks and then combine (e.g., take the weighted average) the output of these networks at minimal extra cost. Hansen and Salamon (1990), among many others, have shown that using such a collective decision strategy improves generalization over a single network. As an initial test, I combined the predictions of all 20 networks in Regent's final population by taking the weighted-average of each network (as determined by its fitness). Simply combining the networks in this fashion produced test-set errors of 7.9% on the RBS domain, 3.8% on the splice-junction domain, and 3.9% on the promoter domain (compared to the errors of 8.2%, 4.1%, and 4.2% respectively of the *best* network kept by Regent). Previous work, though, has shown that an ideal ensemble is one where the networks are both accurate and make their errors on different parts of the input space (Hansen & Salamon, 1990; Krogh & Vedelsby, 1995). Creating such a set of KNNs is the focus of the next chapter.

4.5 Future Work on REGENT

While I present future work specific to Regent in this section, future work within the general framework of my thesis is presented in Chapter 7. Since Regent is searching through many candidate networks, it is important to be able to recognize the networks that are likely to generalize the best. I currently use a validation set; however, MacKay (1992) has shown that a validation set can be a noisy estimate of the true error. Also, as I increase the number of networks searched, Regent may start selecting networks that overfit the validation set. In fact, this may be a possible explanation for the occasional upward trends in test-set error, from both TopGen and Regent, in Figure 19. Future work, then, is to investigate selection methods that do not use a validation set, which would also allow Regent to use all the training instances to train the networks. Such techniques are discussed in more detail in Section 7.2.1.

Often times, there are multiple, even conflicting, theories about a domain. Future work, then, is to investigate ways of using all of these domain theories to seed the initial population. Although the results in Section 4.3.2 show that including randomly generated networks degrades generalization performance, seeding the population with multiple approximately correct theories should not degrade generalization, assuming the networks will have about the same initial correctness. Thus REGENT should be able to naturally combine good parts of multiple theories. Also, for a given domain theory, there are many different but equivalent ways to represent that theory using a set of propositional rules. Each representation leads to a different network topology, and even though each network starts with the same theory, some topologies may be more conducive to neural refinement.

4.6 Wrap-up

In this chapter I presented a new algorithm, REGENT, that uses a specialized genetic algorithm to broaden the types of topologies considered during its search. Experiments indicate that REGENT is able to significantly increase generalization over Chapter 3's TopGen algorithm; hence, my new algorithm is successful in overcoming TopGen's limitation of only searching a small portion of the space of possible network topologies. In doing so, REGENT is able to generate a good solution quickly, by using KBANN, then is able to continually improve this solution as it searches concept space. Therefore, one can view REGENT as an anytime learner, which makes effective use of problem-specific knowledge and available computing cycles. (Versions of this chapter was previously published in Opitz and Shavlik, 1994a, 1994b.)

Chapter 5

Generating Knowledge-Based Neural-Network Ensembles

Many researchers have shown that simply averaging (possibly in a weighted manner) the output of many classifiers can generate more accurate predictions than that of any of the individual classifiers (Clemen, 1989; Granger, 1989; Zhang et al., 1992; Wolpert, 1992; Breiman, 1994). In particular, combining the predictions of separately trained neural networks (commonly referred to as a neural-network *ensemble*) has been demonstrated to be particularly successful (Lincoln & Skrzypek, 1989; Perrone, 1992; Alpaydin, 1993). Both theoretical (Hansen & Salamon, 1990; Krogh & Vedelsby, 1995) and empirical (Hashem et al., 1994; Maclin & Shavlik, 1995) work has shown that a good ensemble is one where the networks are both accurate and make their errors on different regions of the input space. In this chapter, I present an improvement to Chapter 4's REGENT algorithm, called ADDEMUP (Accurate anD Diverse Ensemble-Maker giving United Predictions), that uses genetic algorithms to generate an ensemble of knowledge-based neural networks (KNNs) that are as accurate as possible, while at the same time having minimal overlap on where they make their error.

Previous work on ensembles has either focussed on combining the output of multiple trained networks or only *indirectly* addressed how we should generate a good set of networks. Traditional ensemble techniques generate their networks by randomly trying different topologies, initial weight settings, or parameters settings; or they use only a part of the training set (Hansen & Salamon, 1990; Alpaydin, 1993; Breiman, 1994; Krogh & Vedelsby, 1995; Maclin & Shavlik, 1995). The goal of these approaches is to produce networks that disagree on which inputs they make their errors (I henceforth refer to *diversity* as the measure of this disagreement). I propose instead to actively *search* for a good set of networks. The key idea behind my approach is to use available computer cycles to consider many networks, keeping a subset of the networks that minimizes my objective function consisting of both an accuracy and a diversity term.

ADDEMUP proceeds by first creating an initial set of neural networks, then continually produces new individuals by using the crossover and mutation operators. Currently ADDEMUP uses REGENT to create its networks; though its basic framework can be extended to incorporate other learning algorithms. As I discussed in Chapter 4, one could simply use REGENT's final population as an ensemble; however, ADDEMUP extends this approach by carefully defining the overall fitness of an individual to be a combination of accuracy and diversity. Also, ADDEMUP actively tries to generate good candidates by emphasizing the current population's erroneous examples during backpropagation training. Experiments reported herein demonstrate that ADDEMUP is able to generate a more effective set of networks for its ensemble than is REGENT.

When initializing ADDEMUP's population, one can either randomly create the networks, or create them with KNNs. Creating the initial population with KNNs allows ADDEMUP to have highly correct networks in its ensemble; however, since in this case each network is translated from the same domain theory, we do not expect there to be much disagreement among the networks. The alternative of randomly generating the initial population of network topologies trades off the overall accuracy of each single network for more disagreement between the networks. Experiments in this chapter also show that creating an ensemble of KNNs is more effective than randomly initializing the population for the domains considered in this thesis.

Before presenting the ADDEMUP algorithm, I give a brief introduction and theoretical overview of neural-network ensembles. I then use this theory to motivate the implementation details of ADDEMUP.

5.1 The Importance of an Accurate and Diverse Ensemble

Figure 22 illustrates the basic framework of a neural-network ensemble. Each network in the ensemble (network 1 through network N in this case) is first trained using the training instances. Then, for each example, the predicted output of each of these networks (o_i in Figure 1) is combined to produce the output of the ensemble ($\hat{o}$ in Figure 1). Many researchers (Lincoln & Skrzypek, 1989; Alpaydin, 1993; Hashem et al., 1994; Krogh & Vedelsby, 1995) have demonstrated the effectiveness of combining schemes that are simply the weighted average of the networks (i.e., $\hat{o} = \sum_{i \in N} w_i \cdot o_i$ and $\sum_{i \in N} w_i = 1$), and this is the type of ensemble that I focus on in this chapter. An alternate approach is to have each network only learn a particular subtask (or correction). This has proven to be effective (Jacobs et al., 1991; Baxt, 1992; Nowlan & Sejnowski, 1992; Drucker et al., 1992); however, the literature cited above has shown that the ensemble approach is also effective and general, and thus deserves further investigation.

Combining the output of several networks is useful only if there is disagreement on some inputs. Obviously, combining several identical networks produces no gain. Hansen and Salamon (1990) proved that for a neural-network ensemble, if the average error rate for a pattern is less than 50% and the networks in the ensemble are independent in the production of their errors, the expected error for that pattern can be reduced to zero as the number of networks combined goes to infinity; however, such assumptions rarely hold in practice.

Krogh and Vedelsby (1995) later proved that the ensemble error can be divided into a term measuring the average generalization error of each individual network and a term

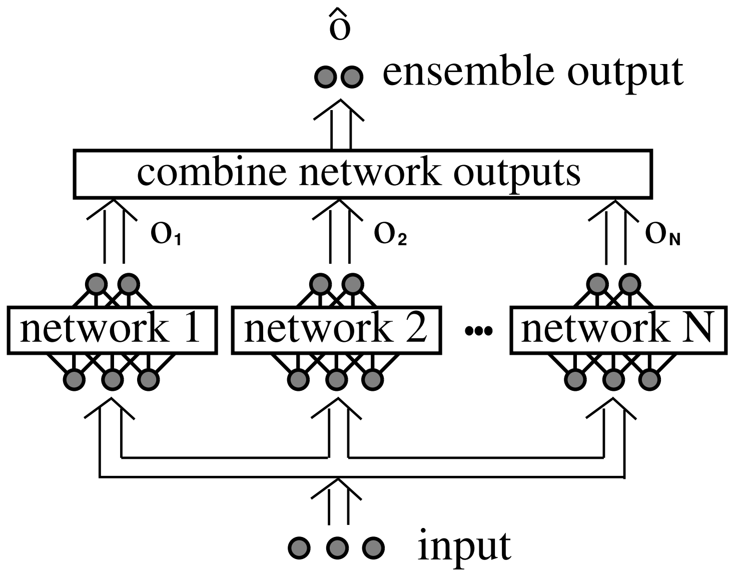


Figure 22: A neural-network ensemble.

called diversity[1] that measures the disagreement among the networks. Formally, they define the diversity term, d_i, of network i on input x to be:

$$d_i(x) = [o_i(x) - \hat{o}(x)]^2. \tag{15}$$

The quadratic error of network i and of the ensemble are:

$$\epsilon_i(x) = [o_i(x) - f(x)]^2, \tag{16}$$

$$e(x) = [\hat{o}(x) - f(x)]^2, \tag{17}$$

respectively, where $f(x)$ is the target value for input x. If we define $\hat{E}$, E_i, and D_i to be the averages, over the input distribution, of $e(x)$, $\epsilon(x)$, and $d(x)$ respectively, then the ensemble's generalization error consists of two distinct portions:

$$\hat{E} = \bar{E} - \bar{D}, \tag{18}$$

where $\bar{E}$ $(= \sum_i w_i E_i)$ is the weighted average of the individual networks' generalization error and $\bar{D}$ $(= \sum_i w_i D_i)$ is the weighted average of the diversity among these networks.

[1]Krogh and Vedelsby referred to this term as *ambiguity*.

What the equation shows then, is that an ideal ensemble consists of highly correct networks that disagree as much as possible. *Creating such a set of networks is the focus of this chapter.*

5.2 The ADDEMUP Algorithm

Table 9 summarizes my algorithm, ADDEMUP, that uses genetic algorithms to generate a set of neural networks that are accurate and diverse in their classifications. ADDEMUP starts by creating and training its initial population of networks. It then creates new networks by using standard genetic operators, such as crossover and mutation. ADDEMUP trains these new individuals, emphasizing examples that are misclassified by the current population, as explained below. It adds new networks to the population and then scores each population member with respect to its classification accuracy and diversity. ADDEMUP normalizes these scores and then defines the fitness of each population member to be:

$$Fitness_i = Accuracy_i + \lambda\ Diversity_i = (1 - E_i) + \lambda\ D_i, \tag{19}$$

where λ defines the tradeoff between accuracy and diversity. Finally, ADDEMUP prunes the population to the N most-fit members, which it then defines to be its current ensemble, then repeats this process.

I define my accuracy term, $1 - E_i$, to be network i's validation-set accuracy (or training-set accuracy if a validation set is not used), and I use Equation 15 over this validation set to calculate my diversity term, D_i. I then separately normalize each term so that the values range from 0 to 1. Normalizing both terms allows λ to have the same meaning across domains.

Since it is not always clear at what value one should set λ, I have therefore developed some rules for automatically adjusting λ. First, I never change λ if the ensemble error $\hat{E}$ is decreasing while I consider new networks; otherwise I change λ if one of following two things happen: (a) population error $\bar{E}$ is not increasing and the population diversity

Table 9: The ADDEMUP algorithm.

GOAL: Genetically create an accurate and diverse ensemble of networks.

1. Create and train the initial population of networks.
2. Until a stopping criterion is reached:
   (a) Use genetic operators to create new networks.
   (b) Train the new networks using Equation 20 and add them to the population.
   (c) Measure the diversity of each network with respect to the current population (see Equation 15).
   (d) Normalize the accuracy scores and the diversity scores of the individual networks.
   (e) Calculate the fitness of each population member (see Equation 19).
   (f) Prune the population to the N fittest networks.
   (g) Adjust λ (see the text for an explanation).
   (h) Report the current population of networks as the ensemble. Combine the output of the networks according to Equation 21.

$\bar{D}$ is decreasing; diversity seems to be under emphasized and I increase λ, or (b) $\bar{E}$ is increasing and $\bar{D}$ is not decreasing; diversity seems to be over-emphasized and I decrease λ. (I started λ at 0.1 for the results in this chapter.)

A useful network to add to an ensemble is one that correctly classifies as many examples as possible, while making its mistakes primarily on examples that most of the current population members correctly classify. I address this during backpropagation training by multiplying the usual error function by a term that measures the combined population error on that example:

$$Cost = \sum_{k \in T} \left| \frac{t(k) - \hat{o}(k)}{\hat{E}} \right|^{\frac{\lambda}{\lambda+1}} [t(k) - a(k)]^2, \tag{20}$$

where $t(k)$ is the target and $a(k)$ is the network activation for example k in the training set T. Notice that since the network is not yet a member of the ensemble, $\hat{o}(k)$ and $\hat{E}$ are not dependent on this network; my new term is thus a constant when calculating the derivatives during backpropagation. I normalize $t(k)-\hat{o}(k)$ by the current ensemble error $\hat{E}$ so that the *average* value of my new term is around 1 regardless of the correctness of the ensemble. This is especially important with highly accurate populations, since $t(k)-\hat{o}(k)$ will be close to 0 for most examples, and the network would only get trained on a few examples. The exponent $\frac{\lambda}{\lambda+1}$ represents the ratio of importance of the diversity term in the fitness function. For instance, if λ is close to 0, diversity is not considered important and the network is trained with the usual cost function; however, if λ is large, diversity is considered important and my new term in the cost function takes on more importance.

I combine the predictions of the networks by taking a weighted sum of the output of each network, where each weight is based on the validation-set accuracy of the network. Thus I define my weights for combining the networks as follows:

$$w_i = \frac{1 - E_i}{\sum_k (1 - E_k)}. \tag{21}$$

While simply averaging the outputs can generate a good composite model (Clemen, 1989), I include the predicted accuracy in my weights since one should believe accurate models more than inaccurate ones. I also tried more complicated models, such as emphasizing confident activations (i.e., activations near 0 or 1), but they did not improve the results on my testbeds.

5.3 Experimental Results

I ran Addemup on Nynex's Max problem set and on the three "reduced" Human Genome Project DNA domains (recognizing *promoters*, *splice-junctions*, and *ribosome-binding sites* - RBS). The reduced DNA datasets are described in Section 3.3.2, and are also used in Chapter 4. As with the previous chapters, I use the reduced datasets to allow the algorithms to consider many networks during their search. Each of these datasets is accompanied by a domain theory; refer to Appendix A for more details on these tasks.

My experiments in this chapter measure the test-set error of ADDEMUP on these tasks. All results presented are from a ten-fold cross validation. Within each fold, I held out 10% of the training instances to be used as a validation set if needed. Each ensemble consists of 20 networks, and the REGENT and ADDEMUP algorithms considered 250 networks during their genetic search.

5.3.1 Generalization Ability of ADDEMUP

In this section, I divide my experiments into two classes: (a) the algorithms randomly create the topology of their networks, and (b) they utilize the domain theory to create their networks. Using KNNs allows one to have in his or her ensemble highly correct networks that tend to agree. The alternative of randomly generating the network topologies thus trades off the overall accuracy of each single network for more disagreement between the networks.

As a point of comparison, I include the results of running Breiman's (1994) *bagging* algorithm both on the KBANN network, and on randomly created, single-hidden-layer networks. I also tried other ensemble approaches, such as randomly creating varying multi-layer network topologies and initial weight settings, but bagging did significantly better on all datasets (by 15-25% on all three DNA domains). Bagging is a "bootstrap" (Efron & Tibshirani, 1993) ensemble method that trains each network in the ensemble with a different partition of the training set. It generates each partition by randomly drawing, with replacement, N examples from the training set, where N is the size of the training set. Breiman (1994) showed that bagging is effective on "unstable" learning algorithms where small changes in the training set result in large changes in predictions. Earlier, Breiman (1984) studied instability, and claimed that neural networks and decision trees are unstable, while k-nearest-neighbor methods are stable.

Table 10: Test-set error from a ten-fold cross validation. Table (a) shows the results from running three learners without the domain theory; Table (b) shows the results of running three learners with the domain theory. Pairwise, one-tailed t-tests indicate that ADDEMUP in Table (b) differs from the other algorithms in both tables at the 95% confidence level, except with REGENT in the splice-junction domain.

Standard neural networks (no domain theory used)				
	Promoters	Splice Junction	RBS	MAX
`best-network`	6.6%	7.8%	10.7%	37.0%
`bagging`	4.6%	4.5%	9.5%	35.7%
ADDEMUP	4.6%	4.9%	9.0%	34.9%

(a)

Knowledge-based neural networks (domain theory used)				
	Promoters	Splice Junction	RBS	MAX
KBANN	6.2%	5.3%	9.4%	35.8%
KBANN-`bagging`	4.2%	4.5%	8.5%	35.6%
REGENT-`best-network`	4.4%	4.1%	8.8%	35.9%
REGENT-`combined`	3.9%	3.9%	8.2%	35.6%
ADDEMUP	2.9%	3.6%	7.5%	34.7%

(b)

Generating Non-KNN Ensembles

Table 10a presents the results from the case where the learners randomly create the topology of their networks (i.e., they do not use the domain theory). Table 10a's first row, `best-network`, results from a single-layer neural network where, for each fold, I trained 20 networks containing between 0 and 100 (uniformly) hidden nodes and used a validation set to choose the best network. The next row, `bagging`, contains the results of applying the bagging algorithm to standard, single-hidden-layer networks, where the number of hidden nodes is randomly set between 0 and 100 for each network. The results confirm Breiman's prediction that bagging would be effective with non-KNNs because of the "instability" of standard neural networks. That is, a slightly different training set can produce large alterations in the predictions of the networks, thereby leading to an effective ensemble.

The bottom row of Table 10a, `ADDEMUP`, contains the results of a run of ADDEMUP where its initial population (of size 20) is randomly generated using REGENT's method for creating networks when no domain theory is present (refer to Section 4.3.2 for more details). Even though ADDEMUP trains each network with the same training set, it still produces results comparable to bagging. The results show that on these domains combining the output of multiple trained networks generalizes better than trying to pick the single-best network. Pairwise, one-tailed t-tests indicate that both `ADDEMUP` and `bagging` differ from `best-network` at the 95% confidence level on all four domains; however, the differences between `ADDEMUP` and `bagging` are not significant.

Generating KNN Ensembles

While the previous section shows the power of a neural-network ensemble, Table 10b demonstrates ADDEMUP's ability to utilize prior knowledge. Again, each ensemble contains 20 networks. The first row of Table 10b contains the generalization results of the KBANN algorithm, while the next row, `KBANN-bagging`, contains the results of the ensemble where each individual network in the ensemble is the KBANN network trained on a different partition of the training set. Even though each of these networks start with the same topology and "large" initial weight settings (i.e., the weights resulting from the domain theory), small changes in the training set still produce significant changes in predictions. Also notice that on all datasets, `KBANN-bagging` is as good as or better than running bagging on randomly generated networks (i.e., `bagging` in Table 10a).

The next two rows result from the REGENT algorithm. The first row, `REGENT-best-network`, contains the results from the single best network output by REGENT, while the next row, `REGENT-combined`, contains the results of simply combining, using Equation 21, the networks in REGENT's final population. Last chapter, I showed the effectiveness of `REGENT-best-network`, and comparing it with the results in Table 10a reaffirms this belief. Notice that simply combining the networks of REGENT's final population (`REGENT-combined`) decreases the test-set error over the single-best network picked by REGENT.

`ADDEMUP`, the final row of Table 10b, mainly differs from `REGENT-combined` in two ways: (a) its fitness function (i.e., Equation 19) takes into account diversity rather than just network accuracy, and (b) it trains new networks by emphasizing the erroneous examples of the current ensemble. Therefore, comparing `ADDEMUP` with `REGENT-combined` helps directly test ADDEMUP's diversity-achieving heuristics. The results in Table 10b show that ADDEMUP is able to generate a more effective ensemble than the other learners.

5.3.2 Lesion Study of ADDEMUP

In this section, I perform a lesion study[2] on ADDEMUP's two main diversity-promoting components: (a) its fitness function (i.e., Equation 19) and (b) its reweighting of each training example based on ensemble error (i.e., Equation 20). Table 11 contains the results for this lesion study. The first row, `REGENT-combined`, is a repeat from Table 10b, where I simply combined the networks of REGENT's final population. The next two rows are "lesions" of ADDEMUP. The first, `ADDEMUP-weighted-examples`, is ADDEMUP with only reweighting the examples during training, while the second, `ADDEMUP-fitness`, is ADDEMUP with only its new fitness function. The final row of the table, `ADDEMUP-both`, is ADDEMUP with both its fitness function and its reweighting mechanism (i.e., a repeat of `ADDEMUP` from Table 10b).

The results show that, while reweighting the examples during training usually helps, ADDEMUP gets most of its generalization power from its fitness function. Reweighting examples during training helps create new networks that make their mistakes on a different part of the input space than the current ensemble; however, these networks may not be as correct as training on each example evenly, and thus may be deleted from the population without an appropriate fitness function that takes into account diversity.

[2] A *lesion* study is a well-defined study aimed at testing one particular aspect of an algorithm.

Table 11: Test-set error on the lesion studies of ADDEMUP. Due to the inherent similarity of each algorithm and the lengthy run-times limiting the number of runs to a ten-fold cross-validation, the difference between the lesions of ADDEMUP is not significant at the 95% confidence level.

	Promoters	Splice Junction	RBS
`REGENT-combined`	3.9%	3.9%	8.2%
`ADDEMUP-weighted-examples`	3.8%	3.8%	7.8%
`ADDEMUP-fitness`	3.1%	3.7%	7.4%
`ADDEMUP-both`	2.9%	3.6%	7.5%

5.4 Discussion of ADDEMUP

The results within each of this chapter's table show that combining the output of multiple trained networks generalizes better than trying to pick the single-best network, verifying the conclusions of previous work (Lincoln & Skrzypek, 1989; Hansen & Salamon, 1990; Mani, 1991; Perrone, 1992; Alpaydin, 1993; Breiman, 1994; Hashem et al., 1994; Krogh & Vedelsby, 1995; Maclin & Shavlik, 1995). When generating KNN ensembles, since every network in the population comes from the same set of rules, we expect each network to be similar. Thus the magnitude of the improvements of the KNN ensembles, especially `KBANN-bagging` and `REGENT-combined`, comes as a bit of a surprise. REGENT, however, does create some diversity during its search to ensure a broad consideration of the concept space (Holland, 1975; Goldberg, 1989). It does this by randomly perturbing the topology of each knowledge-based neural network in the initial population and it also encourages diversity when creating new networks during the search through its mutation operator.

While REGENT *encourages* diversity in its population, it does not *actively* search for a diverse population like ADDEMUP. In fact the single best network produced by ADDEMUP (5.1% error rate on the promoter domain, 5.3% on the splice-junction domain, and 9.1% on the RBS domain) is distinctively worse than REGENT's single best network (4.4%, 4.1%, and 8.8% on the three respective domains). Thus, while too much diversity does not allow the population to find and improve the *single* best network, the results in Table 10b show that more diversity is needed when generating an effective ensemble. There are two main reasons why I think the results of ADDEMUP in Table 10b are especially encouraging: (a) by comparing ADDEMUP with `REGENT-combined`, I explicitly test the quality of my fitness function and demonstrate its effectiveness, and (b) ADDEMUP is able to effectively utilize background knowledge to decrease the error of the individual networks in its ensemble, while still being able to create enough diversity among them so as to improve the overall quality of the ensemble.

5.5 Future Work on ADDEMUP

While I present future work specific to ADDEMUP in this section, Chapter 7 contains the limitations and future plans to the general framework of my thesis. (The work in Chapter 7 includes: new techniques for scoring each network; applying my anytime framework to other types of learning algorithms; translating new types of domain theories other than propositional rules; and finally, extracting humanly understandable rules from the final network, or set of networks, output by my algorithms.)

My first planned extension (specific to ADDEMUP) is to investigate new methods for *creating* networks that are diverse in their classification. While ADDEMUP currently tries to generate such networks by reweighting the error of each example, the lesion study showed that ADDEMUP gets most of its increase in generalization from its fitness function. One alternative I plan to try is the bagging algorithm. Breiman (1994) showed that bagging is effective if the learning algorithm is unstable. Therefore, given the variety of topologies ADDEMUP considers during its search, bagging may be able to further improve ADDEMUP's generalization. I therefore plan to use bootstrapping to assign each new population member's training examples. Moreover, rather than just randomly picking these training instances, I plan to also investigate the utility of more intelligently picking this learning set. For instance, one could emphasize picking examples the current ensemble misclassifies.

Several authors have shown that simply averaging the output of many classifiers improves generalization (Clemen, 1989; Lincoln & Skrzypek, 1989; Mani, 1991); however, many other authors have shown that intelligently setting the combining weights improves generalization even more (Zhang et al., 1992; Alpaydin, 1993; Hashem et al., 1994). Future work, then, is to investigate intelligent methods for setting these weights. Currently, ADDEMUP combines each network in the ensemble by taking the weighted average of the output of each network, where each weight is set to the validation-set accuracy of the network. One approach I plan to implement is a proposed method by Krogh and Vedelsby (1995) that tries to optimally find the settings that minimize the ensemble generalization error in Equation 18. They do this by turning the constraints into a quadratic optimization problem. Thus, while ADDEMUP searches for a *set* of networks that minimize Equation 18, this approach searches for a way to optimally *combine* the set for this equation.

As stated earlier, the framework of ADDEMUP and the theory it builds upon can be applied to any inductive learner, not just neural networks. Future work then, is to investigate applying ADDEMUP to these other learning algorithms as well. With genetic programming (Koza, 1992), for instance, I could translate perturbations of the domain theory into a set of dependency trees (see Figure 6b), then continually create new candidate trees via crossover and mutation. Finally, I would keep the set of trees that are a good fit for my objective function containing both an accuracy and diversity term. By implementing ADDEMUP on a learner that creates its concepts faster than training a neural network, I can better study various issues such as finding good ways to change the tradeoff between accuracy and diversity, investigating the value of normalizing the fitness and diversity terms, and finding the appropriate size of an ensemble.

5.6 Wrap-up

My new algorithm, ADDEMUP, uses genetic algorithms with a novel fitness function to search for a correct and diverse ensemble of knowledge-based neural networks. It does this by using Chapter 4's REGENT algorithm to collect a set of KNNs that best fits an objective function that measures both the accuracy of a network and the disagreement of that network with respect to other members in the set. Experiments demonstrate that my method is able to find an effective set of KNNs for its ensemble. ADDEMUP showed statistically significant improvements in generalization over (a) the single best network seen during the search, (b) a previously proposed ensemble method called bagging (Breiman, 1994) applied to both randomly generated networks and the KBANN network, (c) simply combining REGENT's population, and (d) a run of ADDEMUP where it *randomly* created the network topologies of its initial population. In summary, ADDEMUP is successful in generating a set of KNNs that work well together in producing an accurate prediction.

Chapter 6

Additional Related Work

The work presented in this thesis is related to several research areas. While I have presented background and related work when appropriate throughout this manuscript, I give additional related work here. Although I do not re-address previously reviewed work, I do provide "back-pointers" to these discussions.

I have broken this chapter into three main sections: (a) theory-refinement algorithms, (b) methods for automatically finding suitable neural-network topologies, and (c) techniques for combining the predictions of multiple classifiers. Within each section, I give a brief overview of that area while discussing how that work relates to my algorithms. Throughout, I also give detailed descriptions of the most closely related systems.

6.1 Theory Refinement

The first related-work area I cover is that of theory refinement. I gave a terse introduction of theory refinement in Section 1.2. In this section, I break theory-refinement systems into two categories: (a) those that first convert a domain theory into a neural network and then refine this network, and (b) those that refine a domain theory directly in its initial symbolic ("rule-like") form.

6.1.1 Connectionist Theory-Refinement Techniques

I gave an overview of *connectionist* theory-refinement systems in Section 2.2. As stated in that section, connectionist theory-refinement systems have been developed to refine many types of rule bases. For instance, a number of systems have been proposed for revising certainty-factor rule bases (Fu, 1989; Lacher et al., 1992; Mahoney & Mooney, 1993), finite-state automata (Omlin & Giles, 1992; Maclin & Shavlik, 1993), and mathematical equations (Roscheisen et al., 1991; Scott et al., 1992). Most of these systems work by first translating the domain knowledge into a neural network, then modifying the weights of this resulting network.

Of these systems, KBANN (Towell, 1991) is the most obviously related work. All three of my algorithms use KBANN to translate a propositional domain theory into a network, thus determining the initial topology and weight settings of this network. My systems then go on to refine the topology of this network. While I have already described KBANN in detail in Section 2.2.1, I present three other systems closely related to my work next.

Fletcher and Obradovic's Algorithm

Like my algorithms, Fletcher and Obradovic (1993) also present an approach that adds nodes to a KBANN network. Their system constructs a single layer of nodes, fully connected between the input and output nodes, off to the side of KBANN in a style similar to Chapter 3's Strawman algorithm. They generate new hidden nodes using a variant of Baum and Lang's (1991) constructive algorithm. Baum and Lang's algorithm first divides the feature space with hyperplanes. They find each hyperplane by randomly selecting two points from different classes, then localizing a suitable split between these points. Baum and Lang repeat this process until they generate a fixed number of hyperplanes. Fletcher and Obradovic then map each of Baum and Lang's hyperplanes into one new hidden node, thus defining the weights between the input layer and that hidden node; therefore, the number of hyperplanes generated determines the number of additional hidden nodes they add to the side of KBANN. During training, Fletcher and

Obradovic change only the weights between the new hidden layer and the output layer.

Fletcher and Obradovic's approach differs from Strawman mainly in the training of the network, as well as the fact that they only consider one possible expansion to the network. Their algorithm does not change the weights of the KBANN portion of the network, so modifications to the initial rule base are solely left to the constructed hidden nodes. Thus, their system does not take advantage of KBANN's strength of removing unwanted antecedents and rules from the original rule base.

The DAID Algorithm

The DAID algorithm (Towell & Shavlik, 1992) is an extension to KBANN that uses the domain theory to help *train* the KBANN network. As demonstrated in Section 2.2.2 and in Towell and Shavlik (1994), KBANN is more effective at dropping antecedents than adding them. DAID addresses this limitation by trying to find potentially useful inputs features not mentioned in the domain theory. It does this by backing-up errors to the lowest level of the domain theory, then computing correlations with the features. DAID then increases the weight of the links from the potentially useful input features based on these correlations.

DAID mainly differs from my algorithms in that it does not refine the topology of the KBANN network. Thus, while DAID addresses KBANN's limitation of not effectively adding antecedents, it is still unable to introduce new rules or constructively induce new antecedents (refer to Section 3.1.2 for details how my algorithms are able to do this). DAID will therefore suffer with impoverished domain theories. Also notice that since DAID is an improvement for training KNNs, my algorithms can use DAID to train each network they consider during their search (however, I have not done so).

The RAPTURE Algorithm

RAPTURE (Mahoney & Mooney, 1993) is designed for domain theories containing probabilistic rules. Like most connectionist theory-refinement systems, RAPTURE first translates the domain theory into a neural network, then refines the weights of the network with a modified backpropagation algorithm. An improved version of RAPTURE (Mahoney & Mooney, 1994), however, is able to dynamically refine the topology of its network. It does this by using the UPSTART algorithm (Frean, 1990) to add new nodes to the network.

Aside from being designed for probabilistic rules, RAPTURE differs from the framework of my algorithms in that it adds nodes with the intention of completely learning the training set, not generalizing well. Thus, while RAPTURE hillclimbs until the training set is learned, my algorithms continually search topology space looking for a network that minimizes validation-set error. Also, RAPTURE initially only creates links that are specified in the domain theory, and only explicitly adds links through ID3's (Quinlan, 1986) information-gain metric. My algorithms, on the other hand, fully connect consecutive layers in their networks, allowing each rule the possibility of adding antecedents during training.

6.1.2 Symbolic Theory-Refinement Systems

Additional work related to mine includes purely symbolic theory-refinement systems that modify the domain theory directly in its initial form. Systems such as FOCL (Pazzani & Kibler, 1992) and FORTE (Richards, 1995) are first-order, theory-refinement systems that revise predicate-logic theories. One drawback to these systems is that they currently do not generalize as well as connectionist approaches on many real-world problems, such as the DNA promoter task (Cohen, 1992).

REGAL (Giordana & Saitta, 1993; Giordana et al., 1994) is an example of a first-order, theory-refinement system that uses genetic algorithms to help refine an incomplete or inconsistent domain theory. Their system works by first using an automated theorem prover to recognize unresolved literals in a proof, then uses genetic algorithms to induce corrections to these literals, one at a time. My systems, on the other hand, use genetic algorithms (along with neural learning) to refine the *whole* domain theory at the same time.

Several systems, including mine, have been proposed for refining *propositional* rule bases. Early such approaches could only handle improvements to overly specific theories (Danyluk, 1989) or specializations to overly general theories (Flann & Dietterich, 1989). Later systems such as Rtls (Ginsberg, 1990), Ductor (Cain, 1991), Either (Ourston & Mooney, 1994), Ptr (Koppel et al., 1994), and Tgci (Donoho & Rendell, 1995) were later able to handle both types of refinements. I discuss the Either system as a representative of these propositional systems next.

The EITHER Algorithm

Either has four theory-revision operators: (a) removing antecedents from a rule, (b) adding antecedents to a rule, (c) removing rules from the rule base, and (d) inventing new rules. Either uses these operators to make revisions to the domain theory that correctly classify some of the previously misclassified training examples without undoing any of the correctly classified examples. Either uses inductive learning algorithms to invent new rules; it currently uses ID3 (Quinlan, 1986) as its induction component.

Even though my algorithms add nodes in a manner analogous to how a symbolic system adds antecedents and rules, my underlying learning algorithm is "connectionist." Towell (1991) showed that Kbann was superior to Either on the promoter task, and my algorithms outperform Kbann. Kbann's power on this domain is largely attributed to its ability to make "fine-grain" refinements to the domain theory (Towell, 1991). Because of Either's difficulty on this domain, Baffes and Mooney (1993) presented an extension to it called Neither-MofN that is able to learn M-of-N rules – rules that are true if M of the N antecedents are true. This improvement generated a concept that more closely matches Kbann's generalization performance.

While we want to minimize changes to a theory, we do not want to do it at the expense of accuracy; however, Donoho and Rendell (1995) demonstrate that most existing theory-refinement systems, such as EITHER, suffer in that they are only able to make small, local changes to the domain theory. Thus, when an accurate theory is significantly far in structure from the initial theory, these systems are forced to either become trapped in a local maximum similar to the initial theory, or are forced to drop entire rules and replace them with new rules that are inductively created purely from scratch. My algorithms do not suffer from this in that they translate the theory into the less restricting representation of neural networks (Donoho & Rendell, 1995). Also, REGENT and ADDEMUP are able to further reconfigure the structure of the domain with genetic algorithms.

6.2 Finding Appropriate Network Topologies

My second area of related work covers techniques that attempt to find a good domain-dependent topology by dynamically refining their network's topology during training. Many studies have shown that the generalization ability of a neural network depends on the topology of the network (Baum & Haussler, 1989; Tishby et al., 1989). When trying to find an appropriate topology, one approach is to construct or modify a topology in an incremental fashion. This can be done by starting with too many nodes and weights, then taking some away; or starting with too few and adding some more.

Algorithms that start with too many parameters, then remove nodes and weights during training are called *network-shrinking* algorithms, while algorithms that start with too few parameters, then add nodes and weights during training are called *network-growing* algorithms. I gave an overview of these algorithms in Section 2.1.2. The most obvious difference between my approaches and these algorithms is that mine use domain knowledge and symbolic rule-refinement techniques to help determine their network's topology. A second difference is that these other algorithms restructure their network based solely on training set error, while my approaches use a separate validation set. Finally, my approaches use either beam search or genetic algorithms, rather than hillclimbing, when determining where to add nodes.

Instead of incrementally finding an appropriate topology, one can mount a "richer" search than hillclimbing through the space of topologies. I cover one such approach that involves combining genetic algorithms and neural networks next.

Combining Genetic Algorithms and Neural Networks

Genetic algorithms have been applied to neural networks in two different ways: (a) to optimize the connection weights in a fixed topology, and (b) to optimize the topology of the network. Techniques that solely use genetic algorithms to optimize weights (Whitley & Hanson, 1989; Montana & Davis, 1989) have performed competitively with gradient-based training algorithms; however, one problem with genetic algorithms is their inefficiency in fine-tuned local search, thus the scalability of these methods are in question (Yao, 1993). Kitano (1990b) presents a method that combines genetic algorithms with backpropagation. He does this by using the genetic algorithm to determine the starting weights for a network, which are then refined by backpropagation. Regent and Addemup differ from Kitano's method in that they use a domain theory to help determine each network's starting weights and genetically search, instead, for appropriate network topologies.

Most methods that use genetic algorithms to optimize a network topology are similar to my algorithms in that they also use backpropagation to train each network's weights. Of these methods, many directly encode each link in the network (Miller et al., 1989; Oliker et al., 1992; Schiffmann et al., 1992). These methods are relatively straightforward to implement, and are good at fine tuning small networks (Miller et al., 1989); however, they do not scale well since they require very large matrices to represent all the links in large networks (Yao, 1993). Other techniques (Harp et al., 1989; Kitano, 1990a; Dodd, 1990) only encode the most important features of the network, such as the number of hidden layers, the number of hidden nodes at each layer, etc. These indirect encoding schemes can evolve different sets of parameters along with the network's topology and have been shown to have good scalability (Yao, 1993). Some techniques (Koza & Rice, 1991; Oliker et al., 1992) evolve both the architecture and connection weights at the same time; however, the combination of the two levels of evolution greatly increases the search space.

Regent and Addemup mainly differ from genetic-algorithm-based training methods in that my algorithms are designed for *knowledge-based* neural networks. Thus my algorithms uses domain-specific knowledge and symbolic rule-refinement techniques to aid in determining the network's topology and initial weight setting. Regent also differs in that it does not explicitly encode its networks; rather, in the spirit of Lamarkian evolution, it passes *trained* network weights to offspring. A final difference is that most of these other algorithms restructure their network based solely on training-set error, while my algorithms minimize validation-set error.

6.3 Combining the Predictions of Multiple Networks

A final area related to my work is techniques that save multiple trained networks, then uses a collective decision strategy to classify examples. I break these methods into two groups: (a) neural-network ensembles – methods like Addemup that train each network to learn the entire task, then combine their predictions with a weighting function that is independent of the current input, and (b) mixtures of experts – methods that train each network to learn a specific subtask, then gate the network predictions with a function that depends on the input.

6.3.1 Neural-Network Ensembles

The idea of using an ensemble of networks rather than the single best network has been proposed by several people (Lincoln & Skrzypek, 1989; Hansen & Salamon, 1990; Mani, 1991; Wolpert, 1992; Alpaydin, 1993; Hashem et al., 1994). I presented a framework for these systems along with a theory of what makes an effective ensemble in Section 5.1.

Lincoln and Skrzypek (1989), Mani (1991) and the forecasting literature (Clemen, 1989; Granger, 1989) indicate that a simple averaging of the classifiers generates a very good composite model; however, many later researchers (Wolpert, 1992; Zhang et al., 1992; Perrone, 1992; Alpaydin, 1993; Hashem et al., 1994) have further improved generalization with voting schemes that are complex combinations of each classifier's output.

Hansen and Salamon (1990) showed that generalization increases, when combining multiple neural networks, if the networks are independent in their error. Krogh and Vedelsby (1995) later proved that the ensemble generalization error can be minimized by finding a set of accurate networks that disagree as much as possible. Most approaches, however, fail in that they do not *actively* try to generate such a set of networks. These approaches either *randomly* create their networks (Lincoln & Skrzypek, 1989; Hansen & Salamon, 1990), or *indirectly* try to create diverse networks by training each network with dissimilar learning parameters (Alpaydin, 1993), different network architectures (Hashem et al., 1994), various initial weight settings (Maclin & Shavlik, 1995), or separate partitions of the training set (Breiman, 1994; Krogh & Vedelsby, 1995). Unlike ADDEMUP however, these approaches do not directly address *how* to generate such networks that are optimized for the ensemble as a whole. One method that does actively create members for its ensemble, however, is called *boosting*, which I describe next.

Boosting

Shapire (1990) describes a method, called the boosting algorithm, for converting a learner that is guaranteed to always perform slightly better than random guessing into one that achieves arbitrarily high accuracy. The boosting algorithm first trains a classifier with a set of training examples, then collects an approximately equal number of correct and incorrect examples produced by the first hypothesis and then uses them to generate a second hypothesis. Finally, examples where the first and second hypotheses disagree are used to train a third hypothesis, which breaks ties during classification. This algorithm can be recursively called until a predefined accuracy is reached.

Drucker et al. (1992) applied boosting to neural networks to improve their error rate on a handwritten-digit-recognition task. A problem with the boosting algorithm, however, is that with a finite amount of training examples, unless the first network has very poor performance, there may not be enough examples to generate a second or third training set. For instance, if a KBANN network is trained with 3,000 examples from one of the DNA tasks and it reaches 95% correct, you would need 30,000 examples to find an appropriate training set for the second network. Even more examples would be needed to generate a third training set.

Drucker et al. (1992) addressed this problem with a method that generates new data by deforming existing two-dimensional pixel arrays (e.g., by slightly rotating or shifting the images); however, it is less clear how one would create new data on non-image tasks, such as my DNA domains. In contrast, ADDEMUP trains multiple networks with the whole training set, and keeps the set of networks that best fit its criteria of being accurate and diverse. Therefore, ADDEMUP works well on domains, such as the DNA tasks, that are limited in data.

6.3.2 Mixtures of Local Experts

An alternate approach to the ensemble framework is to train individual networks on a subtask, and to then combine these predictions with a "gating" function that depends on the input. Jacobs et al.'s (1991) adaptive mixtures of local experts, Baxt's (1992) method for identifying myocardial infarction, and Nowlan and Sejnowski's (1992) visual model all train networks to learn specific subtasks. The key idea of these techniques is that a decomposition of the problem into specific subtasks might lead to more efficient representations and training (Hampshire & Waibel, 1989).

Once a problem is broken into subtasks, the resulting solutions need to be combined. Jacobs et al. (1991) propose having the gating function be a network that *learns* how to allocate examples to the experts. Thus the gating network allocates each example to one or more experts, and the backpropagated errors and resulting weight changes are then restricted to these networks (and the gating function). Tresp and Taniguchi (1995) propose a method for determining the gating function *after* the problem has been decomposed and the experts trained. Their gating function is an input-dependent, linear-weighting function that is determined by a combination of the networks' variance on the current input with the likelihood that these networks have seen data "near" that input.

Although the mixtures of experts and ensemble paradigms seem very similar, they are in fact quite distinct from a statistical point of view. The mixtures-of-experts model makes the assumption that a single expert is responsible for each example. In this case, each expert is a model of a region of the input space, and the job of the gating function is to decide from which model the data point originates. Since each network in the ensemble approach learns the whole task rather than just some subtask and thus makes no such mutual exclusivity assumption, ensembles are appropriate when no one model is highly likely to be correct for any one point in our input space.

6.4 Summary of Related Work

In this chapter, I broke previous work related to my thesis into three sections. The first of these was theory refinement. My work differs from most *connectionist* theory-refinement systems in that it is able to effectively modify its topology during training, and the few systems that are able to modify their topology do so in a greedy fashion, whereas I continually consider new candidate topologies using beam search. Besides the choice of inductive-learning component, my work differs from purely symbolic refiners in that it effectively considers changes other than small, local corrections to a domain theory.

I next reviewed methods that dynamically try to find appropriate domain-dependent, neural-network topologies. The framework for my algorithms is distinct in that it is designed for *knowledge-based* neural networks. Also, my systems attempt to minimize validation-set error, rather than training-set error.

The final area of related work I presented involves combining multiple classifiers.

ADDEMUP differs from most of these approaches in that it *actively* searches for a good set of classifiers to combine. Also, I have demonstrated that my approach can effectively utilize prior knowledge.

In summary, my algorithms differ from the above mentioned work in that my approaches are "anytime" learners that continually search, in a non-hillclimbing manner, for improvements to the domain theory. Thus, my work is unique in that is provides a connectionist approach that attempts to effectively utilize available background knowledge and available computer cycles to generate the best concept possible.

Chapter 7

Conclusions

The central point of this thesis is that an *effective* learning system must be able to take advantage of all available resources to improve the quality of the concept it generates. The learning algorithms presented here take a step toward this goal. In particular, they utilize background knowledge by translating a theory describing what is currently known about the domain directly into a knowledge-based neural network (KNN). They then continually refine this network over time, keeping the best network (or set of networks) as their concept.

Before presenting limitations to my thesis, it is necessary to understand its contributions. I therefore present a list of these contributions next. I then discuss current limitations of my algorithms, while presenting future work I plan to pursue that address these limitations. Finally, I give concluding remarks.

7.1 Contributions

Along with presenting and supporting my thesis statement, there were other contributions of this thesis as well. I start by describing the main contribution associated with each learning system, then state other notable contributions presented in this thesis. The following briefly lists each learning system's main contribution:

- *TopGen provides a way to "quickly" alter a KNN during training.* TopGen does this by first using a symbolic interpretation of the trained network to help locate primary errors of the network, then adds new nodes to this network in a manner analogous to adding rules and conjuncts to a symbolic rule base. The lesson learned is that one should overcome weaknesses inherent in neural refinement of a rule base (e.g., inadequate adding of new rules and antecedents to a propositional rule base), by altering a KNN's topology in a manner that exploits the strengths of symbolically refining that same rule base (e.g., suggesting new places to create new rules or antecedents).

- REGENT *searches a "broader" range of KNN topologies than TopGen, and finds better networks if given time to consider many possibilities.* REGENT generates new KNNs by using the genetic operators of crossover and mutation. Studies with REGENT teach us that the crossover operator should try to cross over the rules within a network, rather than just blindly crossing over nodes, and that mutation should be a directed operator that adds new nodes in an intelligent manner.

- ADDEMUP *provides a method for searching for a good "ensemble" of KNNs.* Previous work has shown us that a good ensemble is one where the networks are not only accurate, but are diverse in terms of their classification error. ADDEMUP provides a mechanism for generating such a set of networks. The domain theory helps create accurate networks, while ADDEMUP's fitness function, along with other diversity-promoting techniques, helps create the needed disagreement among these networks.

While the above contributions describe the main goal addressed by each learning algorithm presented in this thesis, there were other notable contributions as well. The following is a list of the most significant of these contributions:

- Carefully controlled experiments on a chess-related domain illustrates the perils of connectionist theory-refinement systems that do not alter their KNN's topology during training. In particular, such systems are restricted in the types of refinements they can make to the theory. Hence when given impoverished domain theories, generalization suffers and the systems must significantly alter their original rules during training (which makes subsequent rule extraction, Towell and Shavlik, 1993, much harder).

- This thesis demonstrates the importance of *anytime learning*. Most current learning algorithms provide only one answer then stop; however, I obtained improved concepts by developing algorithms that are able to produce a good concept quickly, then are able to continually search concept space, reporting the new best concept whenever one is found.

- In theory refinement, it is desirable to be able to correctly classify the examples while deviating from the initial domain theory as little as possible. This is important since the initial theory sums up past experience that may not be present in our current data. In Section 3.3.1, I presented a method for measuring how much the *meaning* of the rules had changed during learning. This corruption is estimated by measuring the error on the set of examples in the test set that the original domain theory classifies correctly. Previous measurements have dealt with syntactic difference (such as counting the number of changed antecedents) rather than the more informative semantic difference.

7.2 Limitations and Future Work

Since a research project is rarely (if ever) "complete," I next present possible future research directions. While I have given suggestions for future work specific to each learning algorithm along the way (Sections 3.5, 4.5, and 5.5), in this section I concentrate on the limitations and suggested improvements to the framework of these systems as a whole.

7.2.1 Network-Scoring Functions

Because my algorithms consider many neural-network topologies during their search, it is important to be able to recognize the networks that are likely to generalize the best to future examples. With this in mind, future work is to develop and test different network-evaluation functions. My current evaluation function is to use a validation set (described in Section 2.1.1); however, validation sets have several drawbacks. First, keeping aside a validation set decreases the number of training instances available for each network. Second, the performance of a validation set can be a noisy approximator of the true error (Weigend et al., 1990; MacKay, 1992). Finally, as I increase the number of networks considered, my algorithms may start picking networks that overfit the validation set. In fact, results in Section 4.3 show that overfitting may start to occur after as few as 500 networks.

To avoid the problem of overfitting the data, a common regression trick is to have a cost function that includes a "smoothness" term along with the error term. The best function, then, will be the smoothest function that also fits the data well. How much importance is placed on either term is called the bias/variance tradeoff (Geman et al., 1992). High belief in the bias (or smoothness) will produce an answer that is smooth, but may not fit the data well. Low belief in the bias, on the other hand, may produce an answer that is complex, but fits the data well.

For neural networks, one can add to the estimated error a smoothness component that is a measure of the complexity of the network. The complexity of the network cannot simply be estimated by counting the number of possible parameters, since there tends to be large duplication in the function of each weight in a network, especially early in the training process (Weigend, 1993). Note that the standard weight-decay terms (Rumelhart et al., 1995) would be insufficient as smoothness terms, since these terms only take into account the distribution of the weights and ignore functional duplication.

Two techniques that try to take into account the *effective* size of the network are Generalized Prediction Error (Moody, 1991) and Bayesian methods (MacKay, 1992). These techniques use only training examples when predicting generalization error. Therefore, my algorithms would be able to use the entire training set to train each considered network. Future work, then, is to investigate utilizing these techniques as appropriate scoring functions for KNNs.

Quinlan and Cameron-Jones (1995) propose adding an additional term to the accuracy and smoothness term that takes into account length of time spent searching. They coin the term "oversearching" to describe the phenomenon where more extensive searching causes lower predictive accuracy. Their claim is that oversearching is orthogonal to overfitting, thus these complexity-based methods alone cannot prevent oversearching. As I increase the number of networks I consider during a search, I too may start oversearching, and thus plan to investigate adding an oversearching term as well.

7.2.2 Anytime Learning

As stated in Section 1.3, it is advantageous for a learner to be able to continually improve its answer over time. In this thesis, I demonstrated three connectionist approaches that are successful in being able to utilize available computer cycles to improve the quality of their answers. Another future direction, then, is to extend other learning algorithms to also have an anytime aspect.

In order to make other learning algorithms anytime in nature, they must be able to consider a wide variety of plausible concepts. Decision trees (Breiman et al., 1984; Quinlan, 1993), for instance, are generally greedy in their selection of picking splits for a node. A wide variety of possible splits may be considered at each node, but once a split is made it generally never gets undone. One possible extension is to introduce backtracking into the splits or use "lookahead search."

Also, different decision-tree algorithms vary on the types of splits they consider, as well as the criterion they use for judging the local effectiveness of each split. As I showed with TopGen, if one has time to consider many possible solutions, then it is important to be able to consider a wide variety of solutions as well. Extrapolating this idea to decision trees, one could generate a population of diverse decision trees, then create new trees through the genetic operators of mutation and crossover. Turney (1995), for instance, proposed a method for genetically optimizing a bit-string encoding a "bias" for generating a decision tree. This bias term contained such parameters as cost of classification error and level of pruning. While Turney (1995) only tried to optimize one tree, a set of accurate and diverse decision trees can be kept for a decision-tree ensemble.

The same idea can be extended to theory-refinement systems as well. As stated in Section 6.1.2, most current symbolic theory-refinement systems are trapped in the initial structure of the domain theory. Extending these algorithms by using the ideas presented in this thesis could help overcome this difficulty.

7.2.3 New Types of Domain Theories

It is often the case that domain theories contain rules other than the propositional rules assumed in this thesis; however, as indicated earlier, many authors have already proposed methods for translating varying types of domain theories into neural networks. These include finite-state automata (Omlin & Giles, 1992; Maclin & Shavlik, 1993), push-down automata (Das et al., 1992), fuzzy-logic rules (Masuoka et al., 1990; Berenji, 1991), probabilistic rules (Fu, 1989; Mahoney & Mooney, 1994), mathematical equations (Roscheisen et al., 1991; Scott et al., 1992), and other types of rules. My analysis of propositional-rule KNNs can easily be converted to these other types of rule bases as well. For instance, these algorithms will also suffer if they do not refine their topology during training. Nodes should be dynamically added in a manner that addresses refinements that are symbolically meaningful and useful, but hard for the network's training algorithm to do. Also, genetic algorithms can refine the network's topology by designing operators that keep intact the dependencies of the rules. Finally, a good ensemble can be generated with an appropriate fitness function that also takes into account diversity among the networks.

One type of language worthy of more investigation is predicate logic. While propositional logic is unable to perform quantification, predicate logic can represent real-world facts as statements and can thus quantify these facts. Currently with my systems, if a domain theory is presented in predicate logic, it must be translated into propositional logic before translation. This translation can be done in finite domains by determining the possible values for each variable in each rule (i.e., as defined by the inputs to the network). However, because a single first-order rule may map to many propositional rules over a set of inputs (variable-bindings), information is lost, such as which nodes and weights correspond to the same first-order rule. One may use weight sharing (Rumelhart et al., 1986) to cluster these weights, however these weights would then be unable to split into different clusters during training.

The information contained in predicate-logic rules can be applied to knowledge-based neural networks in two ways: (a) helping to decide where and how to add hidden nodes, and (b) helping to train a network. Figure 23a shows a predicate-logic rule, its corresponding propositional rules as defined by the network's possible inputs, and finally the corresponding network obtained from the propositional rules. Assume the initial domain theory contains only this predicate-logic rule, and the final concept we are trying to learn, shown in Figure 23b, contains another predicate-logic rule. Notice that when dealing with proposition rules, my algorithms must add and correctly learn three new nodes, even though these additions correspond to adding only one predicate-logic rule. If an algorithm knew that it needed to correct false-negatives for the predicate-logic rule a, it could add all three nodes at once, in the appropriate places. The learning algorithm can cluster weights that correspond to the same variable in a predicate-logic rule using ideas similar to soft-weight sharing (Nowlan & Hinton, 1992), thus allowing weights to dynamically change clusters during training. One remaining difficulty is recursion; however, one can handle recursion by either unrolling the recursive rule to a predefined depth, or by using a "recurrent" network (Hertz et al., 1991) containing feedback links that create loops in the network.

a :– b(X), not c(X–1). Valid arguments for b() and c(): 1, 2, 3.	a :– b(X), not c(X–1). a :– c(X), d(X). Valid arguments for b(), c(), and d(): 1, 2, 3.
a :– b(2), not c(1). a :– b(3), not c(2).	a :– b(2), not c(1). a :– b(3), not c(2). a :– c(1), d(1). a :– c(2), d(2). a :– c(3), d(3).

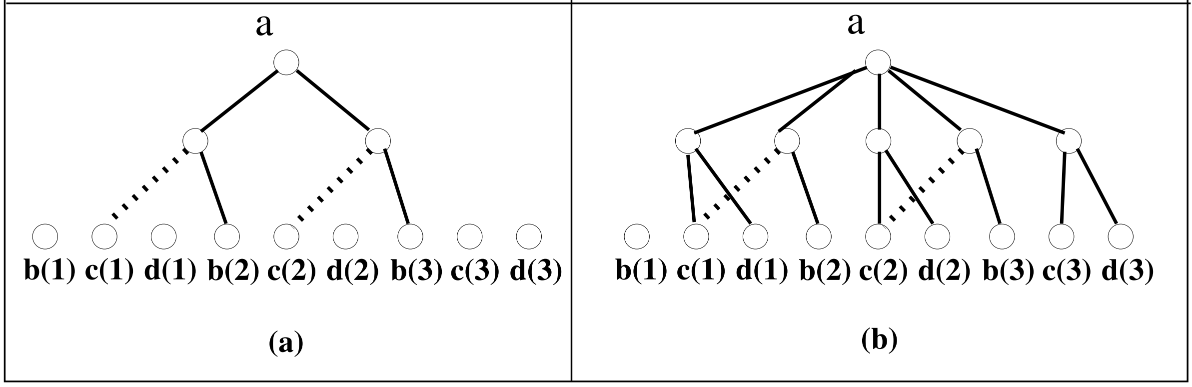


Figure 23: Part (a) shows a predicate-logic rule, the corresponding propositional-logic rules, and the neural network obtained from the propositional rules. Part (b) is the same as part (a) with an additional predicate-logic rule. Each layer is fully-connected to the next layer; however, lightly-weighted links are not shown.

7.2.4 Interpreting Knowledge-Based Neural Networks

The final step of the connectionist theory-refinement framework presented in Section 2.2 is to understand what it is that the networks have learned. Future work, then, is to extract from trained networks human-comprehensible symbolic rules that are in a form similar to the initial rule set (Sestito & Dillon, 1990; Fu, 1991; Towell & Shavlik, 1993). One rule-extraction method designed specifically for KNNs is the NofM algorithm (Towell & Shavlik, 1993). Since Kbann only adds and subtracts antecedents from existing rules, extracting rules from Kbann networks is relatively straight forward; however, it is more difficult to extract rules from neural networks that have not been initialized with a domain theory (Fu, 1991; Craven & Shavlik, 1993). This difficulty results from the large number of nearly uniformly distributed weights that are associated with each node. For this reason, refining the topology of the Kbann network complicates rule extraction.

My first algorithm, TopGen, tries to overcome this complication by adding nodes to the network in a way that is similar to adding rules and conjuncts to the rule base. Towell and Shavlik (1993) showed that KNNs are interpretable because (a) the meaning of their nodes does not significantly shift during training, and (b) the output of each node is nearly binary. Not only does TopGen currently add nodes in a symbolic fashion, it adds them in a fashion that does not violate these two assumptions. I therefore hypothesize that TopGen builds networks that are more interpretable than naive approaches of adding nodes, such as the approach taken by Section 4's Strawman algorithm.

My second algorithm, Regent, may significantly alter the topology of Kbann's network and create large, seemingly complex networks; however, the underlying structure of the networks Regent considers is mostly made up of domain-theory rules. That is, the output of most of the nodes will continue to be nearly binary. Thus, Regent's networks should be more interpretable than standard neural networks.

My final approach, Addemup, saves an ensemble of networks as its final conclusion. Extracting rules directly from the network weights in an ensemble would be a daunting task. Craven and Shavlik (1994b), however, present an approach that views rule extraction as a *learning* problem. The target concept in this case is the function of the network, and since the network can be *queried*, the learning algorithm has access to additional examples outside the training set. Because they treat the network as a black box, their algorithm is applicable to extracting rules from an ensemble as well. One potential drawback of this approach is that the extracted rules may be in a form that is not similar to the initial domain theory; however, one could "prime" the rule-extraction learner with this domain theory and cast the problem as a theory-refinement technique (i.e., the learner could be Pazzani and Kibler's (1992) FOCL, or Ourston and Mooney's (1994) EITHER).

7.3 Concluding Remarks

This dissertation describes three learning systems (TopGen, REGENT, and ADDEMUP) that use the resources of data, available computer cycles, and background knowledge to learn their concepts. These algorithms are unique in that they attempt to be "anytime" learners that continually improve the quality of their concept over time.

Each of these algorithms start by directly translating a theory describing what is currently known about the domain into a neural network. The algorithms then search for refinements to this "knowledge-based" network's topology that produce a more appropriate network (or set of networks). Results show that these algorithms are successful at achieving their intended goal. TopGen finds good "local" refinements to the topology, REGENT finds better "global" changes to this topology, and ADDEMUP finds a "set" of networks that are effective in producing a combined prediction.

These algorithms are but a first step toward realizing the goal of fully being able to utilize all available resources to improve the quality of the concept an inductive learner generates. The hope is that this work is successful in spawning other machine learning researchers to join me in producing anytime-learning, theory-refinement systems.

Appendix A

Experimental Data Sets

I tested my algorithms on six domains: an artificial chess-related domain, four real-world Human Genome problems, and one real-world expert system that diagnoses faults in a telephone loop. The domain theories and datasets for these tasks are explained in this appendix. The domain theories are presented in Prolog notation (Clocksin & Mellish, 1987) that is extended at times (as explained below).

A.1 A Chess Sub-Problem: An Artificial Domain

While real-world domains are clearly useful in exploring the utility of an algorithm, they are difficult to use in closely-controlled studies that examine different aspects of an algorithm. An artificial domain allows me to determine the relationship between the theory provided to the learner and the correct domain theory. My artificial domain is derived from the game of chess and defines board configurations where moving a king one space forward is legal (i.e., the king would not be in check). Figure 24 shows the 4x5 subset of the chess board used. The player is considering moving the king from position $c1$ to position $c2$. Possible pieces include a queen, a rook, a bishop, and a knight for both sides. Each instance is generated by randomly placing a subset of these pieces on the remaining board positions. For example, a queen of the opposing team occupying

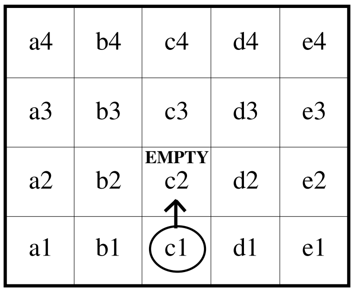


Figure 24: Portion of the chess board covered by the domain theory.

position $b4$, and a bishop from the player's team occupying position $d3$ would comprise one instance of a legal move. The dataset contains 1,500 legal moves and 1,500 illegal ones.

The target domain theory contain 26 propositional rules and is presented in Table 12. The rules are broken into three parts: (a) in check diagonally by the queen or bishop, (b) in check horizontally or vertically by a queen or rook, and (c) in check by a knight. Some rules require a space to be empty between a piece's position and position *c2* (where the player wants to move the king). For instance, if a queen of the opposite color is at position *c4*, then space *c3* must be empty in order for it to be able to put the king in check.

A.2 Finding Genes in DNA Sequences

The next four domains are important subproblems in the computer analysis of DNA sequences. DNA is a linear sequence of four "nucleotides" – adenine, guanine, thymine, and cytosine – that are commonly abbreviated by the letters `A`, `G`, `T`, and `C`. Genes are subsequences of DNA that serve as blueprints for proteins, which in turn provide most of the structure, function, and regulatory mechanisms of cells and are thus the key building blocks of organisms. Researchers are currently sequencing large volumes of DNA; however, biologist are only able to study small sections of DNA at a time. Thus, the Human Genome Project (Cooper, 1994) will produce long runs of DNA that have not

Table 12: Domain theory for the artificial chess problem. The OR in the following rules means that only one of the two antecedents needs to be satisfied in order for the rule to be satisfied. For instance, the first diagonal_check rule can be split into two equivalent rules where one ule contains queen(a4) and the other contains bishop(a4).

```
illegal_move :- diagonal_check.
illegal_move :- parallel_check.
illegal_move :- knight_check.

diagonal_check :- [queen(a4) OR bishop(a4)], opponent(a4), empty(b3).
diagonal_check :- [queen(e4) OR bishop(e4)], opponent(e4), empty(d3).
diagonal_check :- [queen(b3) OR bishop(b3)], opponent(b3).
diagonal_check :- [queen(d3) OR bishop(d3)], opponent(d3).
diagonal_check :- [queen(b1) OR bishop(b1)], opponent(b1).
diagonal_check :- [queen(d1) OR bishop(d1)], opponent(d1).

parallel_check :- [queen(c4) OR rook(c4)], opponent(c4), empty(c3).
parallel_check :- [queen(c3) OR rook(c3)], opponent(c3).
parallel_check :- [queen(a2) OR rook(a2)], opponent(a2), empty(b2).
parallel_check :- [queen(b2) OR rook(b2)], opponent(b2).
parallel_check :- [queen(d2) OR rook(d2)], opponent(d2).
parallel_check :- [queen(e2) OR rook(e2)], opponent(e2), empty(d2).

knight_check :- knight(b4), opponent(b4).
knight_check :- knight(d4), opponent(d4).
knight_check :- knight(a3), opponent(a3).
knight_check :- knight(e3), opponent(e3).
knight_check :- knight(a1), opponent(a1).
knight_check :- knight(e1), opponent(e1).

empty(b3) :- not queen(b3), not rook(b3), not bishop(b3),
             not knight(b3).
empty(c3) :- not queen(c3), not rook(c3), not bishop(c3),
             not knight(c3).
empty(d3) :- not queen(d3), not rook(d3), not bishop(d3),
             not knight(d3).
empty(b2) :- not queen(b2), not rook(b2), not bishop(b2),
             not knight(b2).
empty(d2) :- not queen(d2), not rook(d2), not bishop(d2),
             not knight(d2).
```

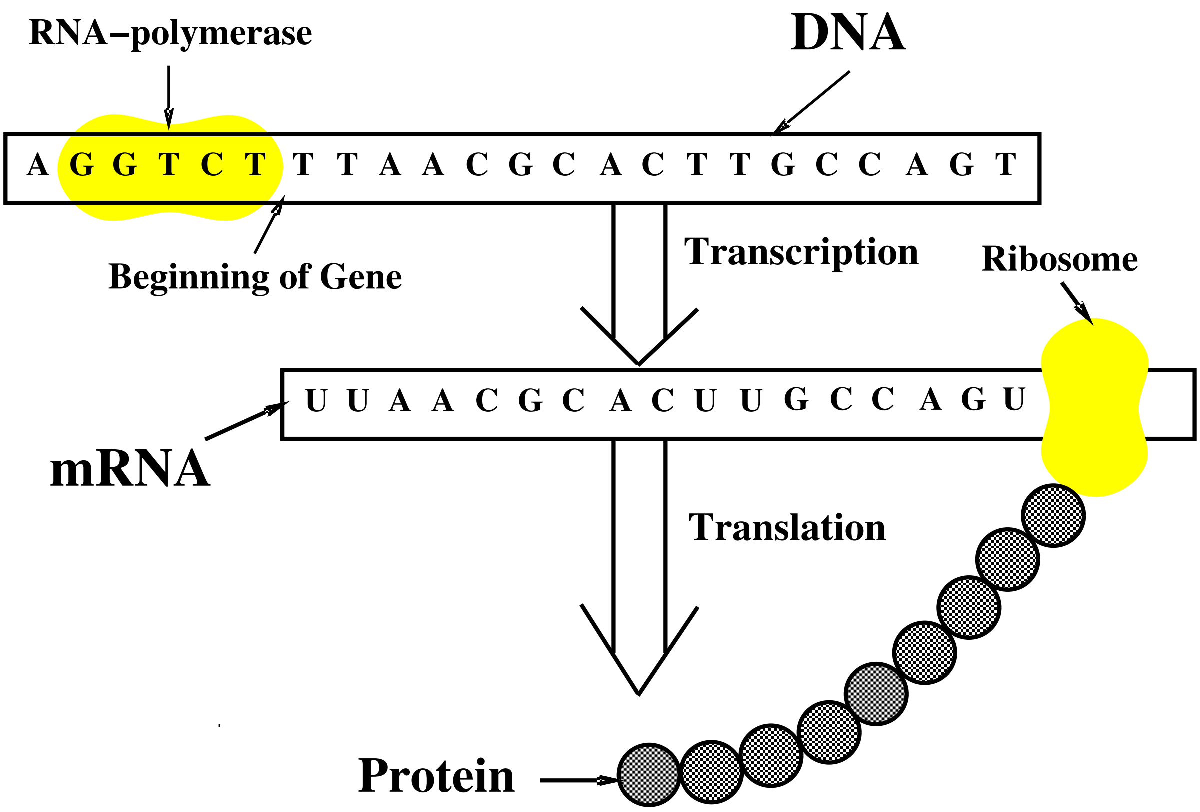


Figure 25: The process of gene expression. This process consists of two phases: (a) *transcription* - when the enzyme RNA-polymerase transcribes a DNA sequence into mRNA, and (b) *translation* - when the ribosome molecule reads the mRNA stand and assembles a protein chain.

---

been analyzed biologically. It is therefore imperative to develop automated techniques that are able to find where genes occur in these unanalyzed sequences.

Figure 25 illustrates the process of gene expression. This process is broken into two phases: *transcription* and *translation*. Transcription happens when the enzyme *RNA-polymerase* transcribes DNA into an RNA molecule called *messenger RNA* (mRNA). The enzyme does this by first binding to a DNA sequence, called a *promoter*, that precedes the gene. It then transcribes the DNA sequence into a similar RNA sequence, except that the nucleotide *thymine* is replaced with the nucleotide *uracil* (U). Translation occurs when the *ribosome* molecule reads the mRNA strand and assembles a protein chain.

One common approach to finding genes is called *search-by-signal* (Stormo, 1987).

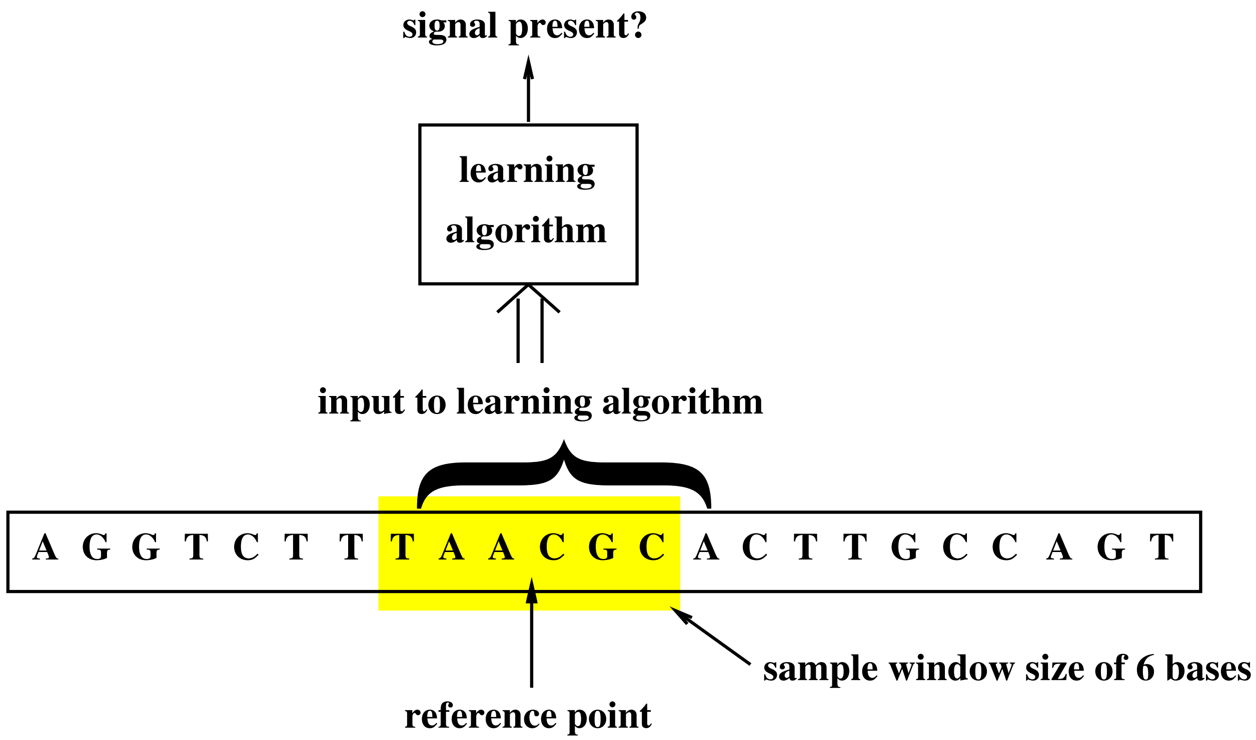


Figure 26: Representing a search-by-signal classification task in a neural network. The input to the neural network is a subsequence (called a *window*) of DNA, and the classification task is to learn when the signal its learning to recognize is present at the reference point.

This approach works by trying to indirectly find genes through specific signals that are associated with gene expression. Not only are these signal detections important for finding genes, they are important in their own right to understand the mechanisms of gene expression. Figure 26 illustrates how I represent the search-by-signal problems in a neural network. The network is given a fixed-length window of DNA with the task of deciding if the desired signal is located at a fixed location in the window. A trained network can then scan a DNA sequence, finding potential points of interest.

The following sections describe four search-by-signal domains that are important in finding genes: (a) promoter sites, (b) splice-junction sites, (c) ribosome-binding sites, and (d) transcription-termination sites. See Craven and Shavlik (1994a) for more details about these tasks. An expert (M. Noordewier) generated all four of the datasets and domain theories from the biological literature. Before I present these domains, however, I describe relevant notation.

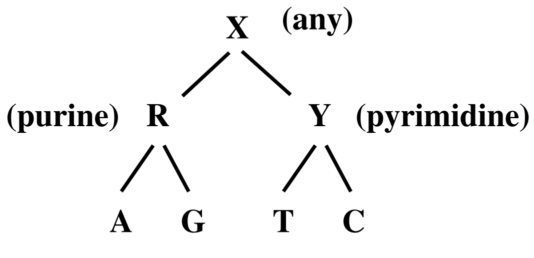
A.2.1 Notation

The domain theories presented in this section use a special notation for specifying locations in a DNA sequence. In this notation, each location is numbered with respect to a fixed, biologically meaningful reference point. Negative numbers are locations preceding the reference point, while positive numbers are locations that follow this point. The following is an example:

Location number:	-3	-2	-1		$+1$	$+2$	$+3$
Sequence:	A	T	A	(reference point)	C	G	A

Note that the biological literature does not use a position zero.

DNA nucleotides are often grouped into the following biologically meaningful hierarchy:



Rules in the following domain theories refer to a string of nucleotides that must occur relative to a location number. For instance, @-39"RA" means that at location -39 there is an A or G, and at location -38 there is an A. Also, in the following theories I follow biological convention and use a W to represent A or T, and an M to represent A or C.

Some domain theories contain M-of-N rules (i.e., a rule's consequent is true if at least M of the rule's N antecedents are satisfied). These rules are of the form:

`consequent :-` M `of (antecedent-list).`

For example, "`T :- 2 of @-39‘AGT’.`" means the consequent, `T`, is considered true if at least two of the three antecedents (i.e., location −39 is an `A`, location −38 is a `G`, and location −37 is a `T`) are satisfied.

A.2.2 Promoter Sites

The first domain is that of recognizing *promoter sites* in a sequence of *E. coli* DNA. As stated above, promoters are short DNA sequences where the RNA-polymerase binds to the DNA. This site is located just "upstream" from where transcription begins; thus locating promoters helps locate genes.

This dataset contains 234 positive examples, and 4,921 negative examples. The reference point in this case is the transcription-initiation site. The input consists of 57 sequential nucleotides, starting at location −50 and ending at location +7. The negative examples are generated from a (putative) promoter-free head of the phage *lambda* that is 4977 bases long.

The approximately correct domain theory shown in Table 13 contains 31 rules that M. Noordewier extracted from the biological literature. Briefly, these rules are characterized by a region rich with `A` and `T` from locations −39 to −35, the sequence `CTTGACA` starting at location −37, and finally another region rich with `A` and `T` directly preceding the reference location. The five promoter rules differ (a) in the type of nucleotides located near position −30 and (b) in the exact location of where the sequence `TATAAT` begins. The domain theory is overly specific; it correctly classifies all the negative examples, but only classifies two of the positive examples correctly. Nonetheless, the rules do capture significant information about promoters. This domain is available at the University of Wisconsin Machine Learning (UW-ML) site via the World Wide Web

Table 13: Domain theory for finding promoters. Refer to Section A.2.1 for an explanation of the notation of the rules as well as the meaning of the letters other than A, G, T, and C. Lines that start with % are comments.

```
% The five promoter rules differ in their spacing between their −35 and −10 regions.
promoter :- bend, minus_35, short_spacer, minus_10_15, melt.
promoter :- bend, minus_35, short_spacer, minus_10_16, melt.
promoter :- bend, minus_35,               minus_10_17, melt.
promoter :- bend, minus_35, long_spacer,  minus_10_18, melt.
promoter :- bend, minus_35, long_spacer,  minus_10_19, melt.

% Look for mostly A's and T's from −39 to −35, and most of CTTGACA at −37.
bend     :- 4 of @-39="WWWWW".
minus_35 :- 6 of @-37="CTTGACA".

% Homo-dinucleotides (RR/YY) pack differently than hetero-dinucleotides (RY/YR).
short_spacer :- 3 of (homonuc1, homonuc2, homonuc3, homonuc4,
                      homonuc5, homonuc6, homonuc7, homonuc8).
long_spacer  :- 3 of (heteronuc1, heteronuc2, heteronuc3, heteronuc4,
                      heteronuc5, heteronuc6, heteronuc7, heteronuc8).

homonuc1   :- @-30="RR".          homonuc2   :- @-29="RR".
homonuc3   :- @-28="RR".          homonuc4   :- @-27="RR".
homonuc5   :- @-30="YY".          homonuc6   :- @-29="YY".
homonuc7   :- @-28="YY".          homonuc8   :- @-27="YY".
heteronuc1 :- @-30="RY".          heteronuc2 :- @-29="RY".
heteronuc3 :- @-28="RY".          heteronuc4 :- @-27="RY".
heteronuc5 :- @-30="YR".          heteronuc6 :- @-29="YR".
heteronuc7 :- @-28="YR".          heteronuc8 :- @-27="YR".

% Look for a close match to the sequence TATAAT near the −10 region.
minus_10_15 :- 5 of @-11="TATAAT".
minus_10_16 :- 5 of @-12="TATAAT".
minus_10_17 :- 5 of @-13="TATAAT".
minus_10_18 :- 5 of @-14="TATAAT".
minus_10_19 :- 5 of @-15="TATAAT".

% Look for mostly A's and T's directly preceeding the site, for thermodynamic reasons.
melt :- 13 of @-15="WWWWWWWWWWWWWWWWWWWWWW".
```

(`http://www.cs.wisc.edu/~shavlik/uwml.html`) or anonymous ftp (`ftp.cs.wisc.edu`, then `cd` to `machine-learning/shavlik-group/datasets`).

Note that this dataset and domain theory are a larger version of the one that appears in Towell (1991) and Towell and Shavlik (1994), which is the original promoter domain found in the UW-ML site and the University of California-Irvine (UCI) Machine Learning Repository. One can access the UCI repository via the web (`http://www.ics.uci.edu/~mlearn/MLRepository.html`) or ftp (`ftp.ics.uci.edu`, then `cd` to `pub/machine-learning-databases`). This testbed was donated to the UCI repository by Noordewier and Shavlik on June 30, 1990. It contains 14 rules and 106 instances, half of which are positive. These positive instances are a subset of the 234 instances used in this thesis.

A.2.3 Splice-Junction Sites

The second domain involves *splice-junction sites*. In *eukaryotic* organisms (i.e., organisms that have cell nuclei), sections of the mRNA are spliced out before translation. Figure 27 illustrates the splicing process. The sections of the mRNA that are translated to proteins are called exons, while the sections that are splice out are referred to as introns. Splice junctions are the boundaries between the introns and exons. There are two types of splice junctions: (a) *acceptors*, which are the intron/exon boundaries, and (b) *donors*, which are the exon/intron boundaries.

The dataset consists of 3,190 examples containing 751 acceptors, 745 donors, knd 1694 negative examples (Towell, 1991). The positive examples are all the documented "split" genes described as complete from the primate gene entries in release 64.1 of Genbank (Burks, 1990).[1] Each example is 60-nucleotides long, with the center being the reference point for the existence of a splice junction; thus, the numbering for the location of each nucleotide range from -30 to $+30$. The negative examples were randomly generated from windows of these sequences containing neither acceptor nor donor sites. Since the

[1]One can access Genbank via the web (`http://genbank.bio.net/`) or anonymous ftp (`genbank.bio.net`).

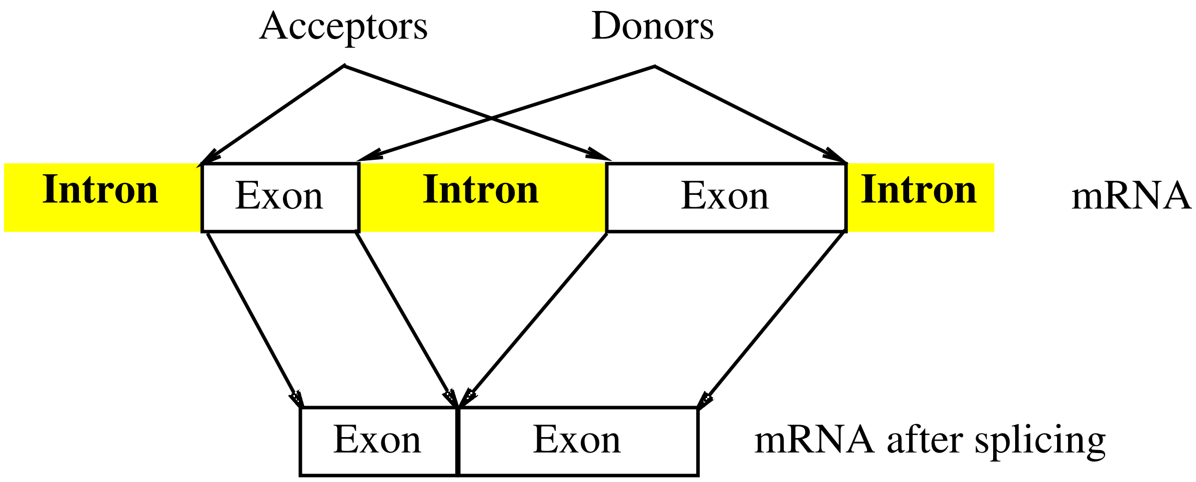


Figure 27: An illustration of the splice-junction process in eukaryotic organisms. Sections of the mRNA called introns ("intervening sequences") are spliced out, while the remaining sections called exons ("expressed sequences") are translated to a gene.

RNA is complementary to the DNA, the input sequences are defined in terms of the corresponding DNA nucleotides, per biological convention.

The domain theory contains 23 rules and is shown in Table 14. As with the previous section, M. Noordewier created this domain theory from the biological literature. Briefly, the domain theory specifies that both acceptors and donors are characterized by a generic sequence near the site, as well as a rule that makes sure that the site is not near the end of a gene. The end of a gene is signified by the sequence of nucleotides `TAA`, `TAG`, and `TGA`, which are called stop codons,[2] Acceptors also contain the constraint that there be a pyramidine-rich region (i.e., a region consisting mainly of `C` and `T`) preceding its site.

The rules correctly classify 60% of the examples; however, since the rules tend to be overly specific, most of the correctly classified examples are negative examples. In fact, the rules only correctly classify 40% of the acceptors and 3% of the donors. The domain theory and dataset was donated by Towell, Noordewier, and Shavlik on January 1, 1992 to the UCI repository, and is also available at the UW-ML site (see the previous section for a description of how to access these sites).

[2] *Codons* are a string of three consecutive nucleotides that encode a single amino acid.

Table 14: Domain theory for finding splice junctions. Refer to Section A.2.1 for an explanation of the notation of the rules as well as the meaning of the letters other than A, G, T, and C.

```
donor :- @+3="MAGGTRAGT", not E/I-stop.

acceptor :- pyramidine-rich, @-3="YAGG", not I/E-stop.

% Make sure that the donor is not near the end of a gene (signified by the stop
% codons TAA, TAG, and TGA).
E/I-stop :- @-3="TAA".  E/I-stop :- @-4="TAA".  E/I-stop :- @-5="TAA".
E/I-stop :- @-3="TAG".  E/I-stop :- @-4="TAG".  E/I-stop :- @-5="TAG".
E/I-stop :- @-3="TGA".  E/I-stop :- @-4="TGA".  E/I-stop :- @-5="TGA".

% Look for a region rich with T's and C's between locations +15 and +6.
pyramidine-rich :- 6 of @+15="YYYYYYYYYY".

% Make sure that the acceptor is not near the end of a gene (signified by the stop
% codons TAA, TAG, and TGA).
I/E-stop :- @+1="TAA".  I/E-stop :- @+2="TAA".  I/E-stop :- @+3="TAA".
I/E-stop :- @+1="TAG".  I/E-stop :- @+2="TAG".  I/E-stop :- @+3="TAG".
I/E-stop :- @+1="TGA".  I/E-stop :- @+2="TGA".  I/E-stop :- @+3="TGA".
```

A.2.4 Ribosome-Binding Sites

The third domain is the task of being able to recognize a *ribosome-binding site* (RBS). As previously shown in Figure 25, RBSs are sites where the mRNA and ribosome bind to each other, and these sites precede where mRNA is translated into proteins. As stated is Section A.2, the ribosome is a complex molecule that reads the mRNA strand to produce the protein's chain of amino acids.

The dataset contains 366 positive examples and 1,511 negative examples. Each instance contains a sequence of 49 nucleotides with the point of reference being a ribosome-binding site. The inputs start at location -25, and since there is no location zero, end at location $+24$. The negative examples are generated from a head of the phage *lambda* that is 1559 bases long and not known to include a ribosome-binding site. With an input window size of 49 bases, 1511 (partially overlapping) negative examples can be generated. As is the case with the splice-junction domain, the input sequences are defined in terms of the DNA nucleotides rather than the corresponding RNA nucleotides.

Table 15: Domain theory for finding ribosome-binding sites. Refer to Section A.2.1 for an explanation of the notation of the rules and Section A.2.4 for a brief explanation of this theory.

```
rbs :- tetranucleotide, start-codon.

tetranucleotide :- agga-region.
tetranucleotide :- gagg-region.

% Look for the start codon ATG from locations +13 to +8.
start-codon :- @+13="ATG".            start-codon :- @+12="ATG".
start-codon :- @+11="ATG".            start-codon :- @+10="ATG".
start-codon :- @+9="ATG".             start-codon :- @+8="ATG".

% Look for the sequence AGGA near the reference point.
agga-region :- @+2="AGGA".            agga-region :- @+1="AGGA".
agga-region :- @-1="AGGA".            agga-region :- @-2="AGGA".

% Look for the sequence GAGG near the reference point.
gagg-region :- @+2="GAGG".            gagg-region :- @+1="GAGG".
gagg-region :- @-1="GAGG".            gagg-region :- @-2="GAGG".
```

Table 15 shows the domain theory, extracted from the biological literature by M. Noordewier. It contains 17 rules which say that a ribosome-binding site contains two parts: (a) either the sequence AGGA or the sequence GAGG near the site, and (b) the start codon ATG beginning 8 to 13 nucleotides before the site. The rules correctly classify about 41% of the positive, and 99% of the negative instances (87.3% overall accuracy). This domain theory and dataset are also available at the UW-ML site. (See Section A.2.2 for details on how to access this site.)

A.2.5 Transcription-Termination Sites

The final domain is recognizing *transcription-termination sites*. Both the transcription and translation processes contain sites where the synthesis of either RNA or proteins is stopped. The translation-termination sites contain specific, known codons (called stop codons) that signal the ribosome to release the mRNA chain; finding these sites is therefore trivial. Finding the short DNA sequences that cause the transcription process

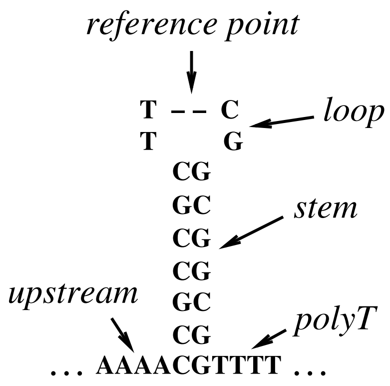


Figure 28: Illustration of a valid transcription-terminator site according to the domain theory in Tables 16 and 17.

---

to terminate, however, is reasonably complex and only partially understood. As with the previous three DNA domains, M. Noordewier also created this section's dataset and domain theory.

The dataset contains 142 positive examples and 5,178 negative examples. The positive examples where taken from the file "gbbct.seq" in release 60 of Genbank (Burks, 1990). The sequences are 50 nucleotides long, spanning from location -20 to location $+30$. The point of reference in this case is transcription-termination sites. The negative examples were generated from sections of a `lambda` DNA (sequence locations 1-1505 and 44,780-48,502 of Genbank entry LAMCG) known not to contain transcription-termination sites.

Tables 16 and 17 show the 61 rules that make up the domain theory. According to this theory, there are four main parts that make up a transcription-termination site: (a) a region rich with `A` and `T` upstream from the "stem" of the terminator, (b) a stem of `GC`-rich self-complimentary nucleotides of length about seven, (c) a "loop" centered about the termination site that is either four or five nucleotides long, and (d) a sequence of thymines (`T`) following the stem. Figure 28 illustrates this concept.

Like the promoter theory, this domain theory is also too specific; however, in this case

Table 16: Part 1 of the domain theory for finding transcription-termination sites; see Table 17 for the remainder of this theory. Section A.2.1 contains an explanation of the notation of the rules, as well as the meaning of the letters other than A, G, T, and C. Lines that start with % are comments.

```
% A terminator is assumed to consist of four regions: (a) an upstream region, (b) a
% stem, (c) a loop, and (d) a run of T. The loop consists of four to five nucleotides.
% While a loop of five nucleotides does not require special constraints, four nucleotides
% is not favorable for a loop and does require some constraints.
terminator :- upstream, stem-odd, polyT.
terminator :- upstream, stem-even, loop-even, polyT.

% This rule encodes the notion that a longer stem is more stable, and the fact that
% the shorter the stem, the higher the GC content should be. The consequent of
% the rule, stem-odd, is true if the sum of the "weighted" true antecedents is greater
% than 6. For instance, stem-odd-1 has a weight of 1.3.
stem-odd :- weighted(6, [stem-odd-1 1.3] [stem-odd-2 1.2]
                        [stem-odd-3 1.1] [stem-odd-4 1.0]
                        [stem-odd-5 0.9] [stem-odd-6 0.8]
                        [stem-odd-7 0.7]).

% The top of the stem strongly favors a G-C base-pair, while the rest can contain
% a G-C pair or an A-T pair. Since the lenth of the loop is odd, location -n pairs
% with location +(n + 1). For example, location -3 pairs with location +4.
stem-odd-1 :- @-3="G", @+4="C".         stem-odd-1 :- @-3="C", @+4="G".
stem-odd-2 :- @-4="G", @+5="C".         stem-odd-2 :- @-4="C", @+5="G".
stem-odd-3 :- @-5="A", @+6="T".         stem-odd-3 :- @-5="G", @+6="C".
stem-odd-3 :- @-5="C", @+6="G".         stem-odd-3 :- @-5="T", @+6="A".
stem-odd-4 :- @-6="A", @+7="T".         stem-odd-4 :- @-6="G", @+7="C".
stem-odd-4 :- @-6="C", @+7="G".         stem-odd-4 :- @-6="T", @+7="A".
stem-odd-5 :- @-7="G", @+8="C".         stem-odd-5 :- @-7="C", @+8="G".
stem-odd-5 :- @-7="A", @+8="T".         stem-odd-5 :- @-7="T", @+8="A".
stem-odd-6 :- @-8="G", @+9="C".         stem-odd-6 :- @-8="C", @+9="G".
stem-odd-6 :- @-8="A", @+9="T".         stem-odd-6 :- @-8="T", @+9="A".
stem-odd-7 :- @-9="G", @+10="C".        stem-odd-7 :- @-9="C", @+10="G".
stem-odd-7 :- @-9="A", @+10="T".        stem-odd-7 :- @-9="T", @+10="A".
```

Table 17: Part 2 of the domain theory for finding transcription-termination sites; Table 16 contains the first half of this theory.

```
% A region upstream from the site should be rich with A's and T's.
upstream :- 7 of @-17="WWWWWWWWWWW".

% A short loop of four nucleotides prefers one of the following sequences.
loop-even :- @-2="TTCG".              loop-even :- @-2="GAAA".
loop-even :- @-2="TGCG".              loop-even :- @-2="TTTT".

% As in Table 16, this rule encodes the notion that a longer stem is more stable. Like
% before, the consequent of this rule, stem-even, is true if the sum of the "weighted"
% true antecedents is greater than 6.
stem-even :- weighted(6, [stem-even-1 1.3] [stem-even-2 1.2]
                         [stem-even-3 1.1] [stem-even-4 1.0]
                         [stem-even-5 0.9] [stem-even-6 0.8]
                         [stem-even-7 0.7]).

% The top of the stem strongly favors a G-C base-pair, while the rest can contain
% a G-C pair, or an A-T pair. Since the lenth of the loop is even, location -n pairs
% with location +n. For example, location -3 pairs with location +3.
stem-even-1 :- @-3="G", @+3="C".      stem-even-1 :- @-3="C", @+3="G".
stem-even-2 :- @-4="G", @+4="C".      stem-even-2 :- @-4="C", @+4="G".
stem-even-3 :- @-5="A", @+5="T".      stem-even-3 :- @-5="G", @+5="C".
stem-even-3 :- @-5="C", @+5="G".      stem-even-3 :- @-5="T", @+5="A".
stem-even-4 :- @-6="A", @+6="T".      stem-even-4 :- @-6="G", @+6="C".
stem-even-4 :- @-6="C", @+6="G".      stem-even-4 :- @-6="T", @+6="A".
stem-even-5 :- @-7="G", @+7="C".      stem-even-5 :- @-7="C", @+7="G".
stem-even-5 :- @-7="A", @+7="T".      stem-even-5 :- @-7="T", @+7="A".
stem-even-6 :- @-8="G", @+8="C".      stem-even-6 :- @-8="C", @+8="G".
stem-even-6 :- @-8="A", @+8="T".      stem-even-6 :- @-8="T", @+8="A".
stem-even-7 :- @-9="G", @+9="C".      stem-even-7 :- @-9="C", @+9="G".
stem-even-7 :- @-8="A", @+8="T".      stem-even-7 :- @-8="T", @+8="A".

% The sequence following the stem should consist of a run of thymine (T).
polyT :- @+9="TTTTT".
polyT :- @+10="TTTTT".
polyT :- @+11="TTTTT".
```

no positive examples are correctly classified. Nonetheless, as was the case with the promoters, this theory still contains useful information about the presence of a transcription-termination site.

A.3 NYNEX's MAX System: Finding Errors in Telephone Lines

The last domain I used is NYNEX's MAX system (Rabinowitz et al., 1991). The following discussion is derived from Rabinowitz et al. (1991) and Provost and Danyluk (1995). MAX is an expert system that was designed by NYNEX to diagnose the location of customer-reported telephone problems. Figure 29 illustrates MAX's task. When a customer calls with a phone-service problem, a representative invokes a mechanized loop test (MLT) to create an electronic profile of the voltages and resistances in the loop between the customer's telephone and the central office. Next, a primitive rule-based system, called the screening decision unit (SDU), receives a two-character summary of the MLT, called a *vercode*, and makes a decision based on this vercode. In general, the vercode does not provide enough information to make an accurate decision, so most of the cases are forwarded to a maintenance administrator (MA). The MA evaluates the MLT and vercode readings, then decides how the company should dispatch the trouble.

MAX's job is to emulate the job of a human MA. MAX receives as input the results of the MLT, the vercode, knowledge about the customer's line, and general knowledge about the equipment. Based on this information, MAX makes one of five possible diagnoses: (a) dispatch repair technician to distribution facilities, (b) dispatch repair technician to cable facilities, (c) dispatch repair technician to central office, (d) request retest of the MLT results, or (e) defer to a human MA. The goals of MAX are to shorten the time to diagnose and fix a trouble; have fewer "handoffs" from person to person when analyzing and repairing a customer trouble; reduce the number of incorrect dispatches; and finally, reduce the heavy workload of human MAs at certain sites.

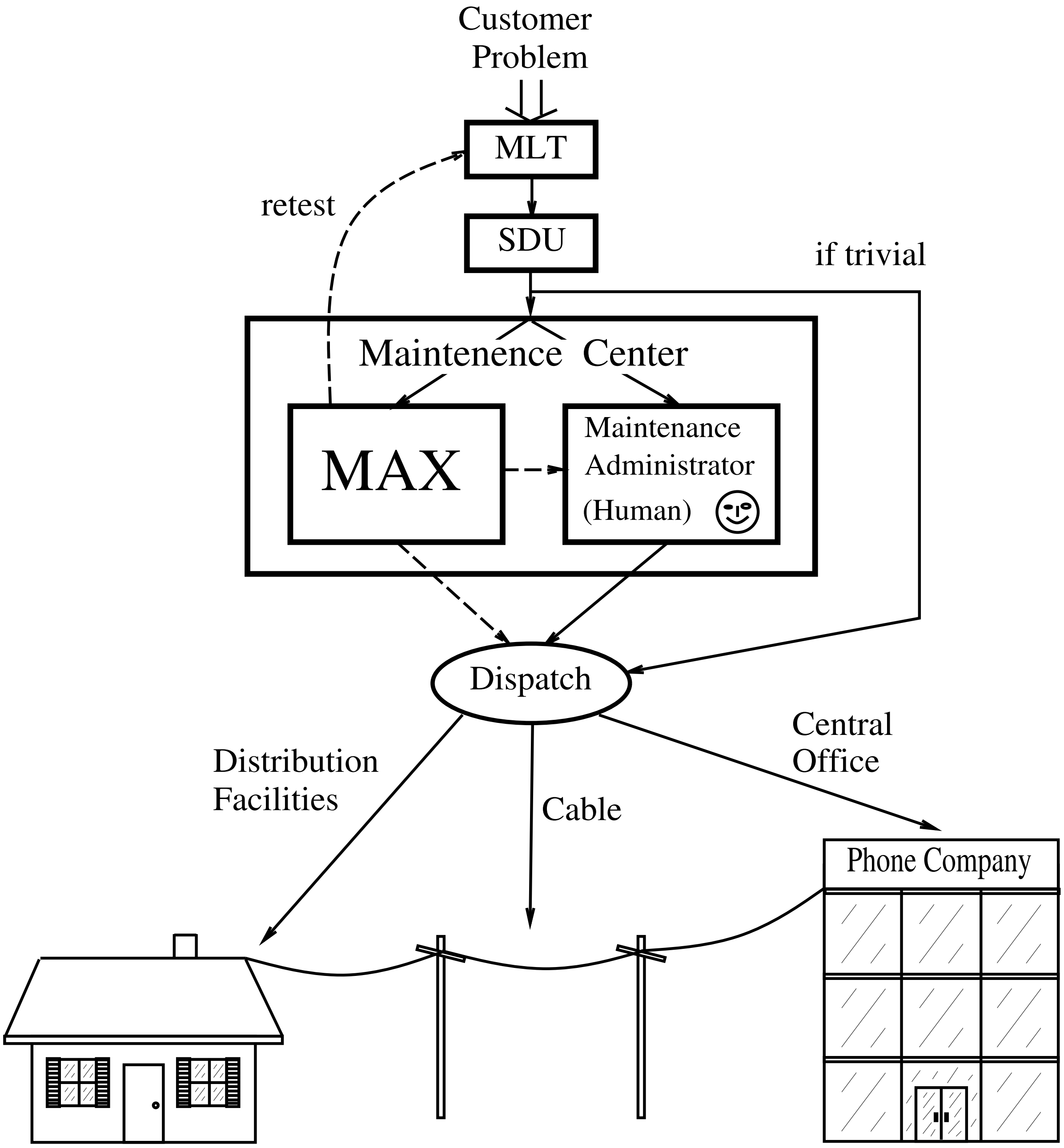


Figure 29: Flowchart of Nynex's handling of customer's troubles.

Max is currently running in over 55 maintenance centers; however, at each center, a large number of parameters must be set to customize Max's knowledge base to that center. Setting these parameters can be difficult. In addition, the job of the MA is made more difficult each year based on the addition of new and non-standard equipment. Thus the knowledge base of Max must be periodically updated. Being able to refine Max's knowledge base with a current set of site-specific examples would greatly simplify both these difficulties.

Max screens over 10,000 phone-service troubles each day (about 38% of all troubles). In fact, some Nynex centers send Max over 50% of their troubles. Given the large number of troubles handled by Max, coupled with the fact that each dispatch usually takes a highly trained worker at least one hour, even a small improvement in Max's accuracy can be extremely valuable. In fact, for every 1% reduction in dispatch error rate, $3 million annually is saved by the company (Provost & Danyluk, 1995).

Max's current knowledge base consists of about 75 ART-Lisp rules, many of which involve inequalities. I converted these rules into an equivalent set of 95 propositional, Prolog-style rules. Due to proprietary reasons, however, I am unable to present this rule base in my thesis.

The data set I use in this thesis is the "cleaned-up" version of dispatching customer problems used by Provost and Danyluk (1995). Any instance where the correct dispatch was deemed as being highly questionable was removed from a larger data set, and the remaining examples make up this data set. The "target" dispatch for each example is the trouble reported by the repair technician. The goal of this domain is to learn proper dispatching of customer problems, and thus one does not have the option of requesting a retest, or deferring to a human MA. This data set consists of 2,686 examples: 918 dispatches to the distribution facilities, 1,412 dispatches to the cable facilities, and 356 dispatches to the central office. Each example consists of 13 input features. I did not include the vercode as an input feature, since it is poorly understood and thus does not allow the possibility of extracting useful information describing the concept description.

Bibliography

Ackley, D. (1987). *A Connectionist Machine for Genetic Hillclimbing*. Kluwer, Norwell, MA.

Ackley, D. & Littman, M. (1994). A case for Lamarckian evolution. In Langton, C., editor, *Artificial Life III*, (pp. 3–10), Redwood City, CA. Addison-Wesley.

Alpaydin, E. (1993). Multiple networks for function learning. In *Proceedings of the 1993 IEEE International Conference on Neural Networks (volume I)*, (pp. 27–32), San Fransisco.

Ash, T. (1989). Dynamic node creation in backpropagation networks. *Connection Science*, 1:365–376.

Atlas, L., Cole, R., Connor, J., El-Sharkawi, M., Marks II, R., Muthusamy, Y., & Barnard, E. (1989). Performance comparisons between backpropagation networks and classification trees on three real-world applications. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 622–629), San Mateo, CA. Morgan Kaufmann.

Baffes, P. & Mooney, R. (1993). Symbolic revision of theories with M-of-N rules. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, (pp. 1135–1140), Chambery, France.

Baum, E. & Haussler, D. (1989). What size net gives valid generalization? *Neural Computation*, 1:151–160.

Baum, E. & Lang, K. (1991). Constructing hidden units using examples and queries. In Lippmann, R., Moody, J., & Touretzky, D., editors, *Advances in Neural Information Processing Systems (volume 3)*, (pp. 904–910), San Mateo, CA. Morgan Kaufmann.

Baxt, W. (1992). Improving the accuracy of an artificial neural network using multiple differently trained networks. *Neural Computation*, 4:772–780.

Berenji, H. (1991). Refinement of approximate reasoning-based controllers by reinforcement learning. In *Proceedings of the Eighth International Machine Learning Workshop*, (pp. 475–479), Evanston, IL.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. (1987). Occam's razor. *Information Processing Letters*, 24:377–380.

Bookman, L. & Sun, R. (1993). Integrating neural and symbolic processes. *Connection Science*, 5:203–204.

Breiman, L. (1994). *Bagging predictors.* Technical Report 421, Department of Statistics, University of California, Berkeley.

Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and Regression Trees.* Wadsworth and Brooks, Monterey, CA.

Bryson, A. & Ho, Y. (1969). *Applied Optimal Control.* Blaisdell, New York.

Burks, C. (1990). Genbank: Current status and future directions. In Doolittle, R. F., editor, *Methods in Enzymology (volume 183).* Academic Press.

Cain, T. (1991). The DUCTOR: A theory revision system for propositional domains. In *Proceedings of the Eighth International Machine Learning Workshop*, (pp. 485–489), Evanston, IL.

Chauvin, Y. (1988). A back-propagation algorithm with optimal use of hidden units. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 1)*, (pp. 519–525), San Mateo, CA. Morgan Kaufmann.

Clemen, R. (1989). Combining forecasts: A review and annotated bibliography. *International Journal of Forecasting*, 5:559–583.

Clocksin, W. & Mellish, C. (1987). *Programming in Prolog.* Springer-Verlag, New York.

Cohen, W. (1992). Compiling prior knowledge into an explicit bias. In *Proceedings of the Ninth International Conference on Machine Learning*, (pp. 102–110), Aberdeen, Scotland.

Cooper, N. G., editor (1994). *The Human Genome Project: Deciphering the Blueprint of Heredity.* University Science Books, Mill Valley, CA.

Craven, M. & Shavlik, J. (1993). Learning symbolic rules using artificial neural networks. In *Proceedings of the Tenth International Conference on Machine Learning*, (pp. 73–80), Amherst, MA.

Craven, M. & Shavlik, J. (1994a). Machine learning approaches to gene recognition. *IEEE Expert*, 9:2–10.

Craven, M. & Shavlik, J. (1994b). Using sampling and queries to extract rules from trained neural networks. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 37–45), New Brunswick, NJ.

Danyluk, A. (1989). Finding new rules for incomplete theories: Explicit biases for induction with contextual information. In *Proceedings of the Sixth International Workshop on Machine Learning*, (pp. 34–36), Ithaca, NY.

Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection*. John Murray, London.

Das, A., Giles, C., & Sun, G. (1992). Using prior knowledge in an NNPDA to learn context-free languages. In Hanson, S., Cowan, J., & Giles, C., editors, *Advances in Neural Information Processing Systems (volume 5)*, (pp. 65–72), San Mateo, CA. Morgan Kaufmann.

Dean, T. & Boddy, M. (1988). An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, (pp. 49–54), St. Paul, MN.

Dodd, N. (1990). Optimization of network structure using genetic techniques. In *Proceedings of the IEEE International Joint Conference on Neural Networks (volume III)*, (pp. 965–970), Paris.

Donoho, S. & Rendell, L. (1995). Rerepresenting and restructuring domain theories: A constructive induction approach. *Journal of Artificial Intelligence Research*, 2:411–446.

Drucker, H., Schapire, R., & Simard, P. (1992). Improving performance in neural networks using a boosting algorithm. In Hanson, J., Cowan, J., & Giles, C., editors, *Advances in Neural Information Processing Systems (volume 5)*, (pp. 42–49), Palo Alto, CA. Morgan Kaufmann.

Efron, B. & Tibshirani, R. (1993). *An introduction to the Bootstrap*. Chapman and Hall, New York.

Fahlman, S. & Lebiere, C. (1989). The cascade-correlation learning architecture. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 524–532), San Mateo, CA. Morgan Kaufmann.

Farmer, J. & Belin, A. (1992). Artificial life: The coming evolution. In Langton, C., Taylor, C., Farmer, J. D., & Rasmussen, S., editors, *Artificial Life II*, (pp. 815–840), Redwood City, CA. Addison-Wesley.

Fisher, D. & McKusick, K. (1989). An empirical comparison of ID3 and backpropagation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, (pp. 788–793), Detroit, MI.

Flann, N. & Dietterich, T. (1989). A study of explanation-based methods for inductive learning. *Machine Learning*, 4:187–226.

Fletcher, J. & Obradovic, Z. (1993). Combining prior symbolic knowledge and constructive neural network learning. *Connection Science*, 5:365–375.

Fletcher, R. (1987). *Practical Methods of Optimization*. John Wiley and Sons, Chichester, second edition.

Forrest, S. & Mitchell, M. (1993). What makes a problem hard for a genetic algorithm? Some anomalous results and their explanation. *Machine Learning*, 13:285–319.

Frean, M. (1990). The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2:198–209.

Fu, L. (1989). Integration of neural heuristics into knowledge-based inference. *Connection Science*, 1:325–340.

Fu, L. (1991). Rule learning by searching on adapted nets. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, (pp. 590–595), Anaheim, CA.

Geman, S., Bienenstock, E., & Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–58.

Ginsberg, A. (1990). Theory reduction, theory revision, and retranslation. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, (pp. 777–782), Boston, MA.

Giordana, A. & Saitta, L. (1993). REGAL: An integrated system for relations using genetic algorithms. In *Proceedings of the Second International Workshop on Multistrategy Learning*, (pp. 234–249), Harpers Ferry, WV.

Giordana, A., Saitta, L., & Zini, F. (1994). Learning disjunctive concepts by means of genetic algorithms. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 96–104), New Brunswick, NJ.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA.

Granger, C. (1989). Combining forecasts: Twenty years later. *Journal of Forecasting*, 8:167–173.

Grefenstette, J. & Ramsey, C. (1992). An approach to anytime learning. In *Proceedings of the Ninth International Conference on Machine Learning*, (pp. 189–195), Aberdeen, Scotland. Morgan Kaufmann.

Hampshire, J. & Waibel, A. (1989). *The meta-pi network: Building distributed knowledge representations for robust pattern recognition*. Technical Report TR CMU-CS-89-166, CMU, Pittsburgh, PA.

Hansen, L. & Salamon, P. (1990). Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:993–1001.

Hanson, S. (1989). Meiosis networks. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 533–541), San Mateo, CA. Morgan Kaufmann.

Hanson, S. & Pratt, L. (1988). Comparing biases for minimal network construction with back-propagation. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 1)*, (pp. 177–185), San Mateo, CA. Morgan Kaufmann.

Harp, S., Samad, T., & Guha, A. (1989). Designing application-specific neural networks using the genetic algorithm. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 447–454), San Mateo, CA. Morgan Kaufmann.

Hashem, S., Schmeiser, B., & Yih, Y. (1994). Optimal linear combinations of neural networks: An overview. In *Proceedings of the 1994 IEEE International Conference on Neural Networks*, Orlando, FL.

Hassibi, B. & Stork, D. (1992). Second order derivatives for network pruning: Optimal brain surgeon. In Hanson, S., Cowan, J., & Giles, C., editors, *Advances in Neural Information Processing Systems (volume 5)*, (pp. 164–171), San Mateo, CA. Morgan Kaufmann.

Hertz, J., Krogh, A., & Palmer, R. (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley, Redwood City, CA.

Hinton, G. (1986). Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, (pp. 1–12), Amherst, MA.

Hinton, G. & Nowlan, S. (1987). How learning can guide evolution. *Complex Systems*, 1:495–502.

Holder, L. (1991). *Maintaining the Utility of Learned Knowledge Using Model-Based Control*. PhD thesis, Computer Science Department, University of Illinois at Urbana-Champaign.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.

Jacobs, R., Jordan, M., Nowlan, S., & Hinton, G. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3:79–87.

Kitano, H. (1990a). Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476.

Kitano, H. (1990b). Empirical studies on the speed of convergence of neural network training using genetic algorithms. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, (pp. 789–795), Boston, MA.

Koppel, M., Feldman, R., & Segre, A. (1994). Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence Research*, 1:159–208.

Koza, J. (1992). *Genetic Programming*. MIT Press, Cambridge, MA.

Koza, J. & Rice, J. (1991). Genetic generation of both the weights and architectures for a neural network. In *International Joint Conference on Neural Networks (volume 2)*, (pp. 397–404), Seattle, WA.

Krogh, A. & Vedelsby, J. (1995). Neural network ensembles, cross validation, and active learning. In Tesauro, G., Touretzky, D., & Leen, T., editors, *Advances in Neural Information Processing Systems (volume 7)*, Cambridge, MA. MIT Press.

Lacher, R., Hruska, S., & Kuncicky, D. (1992). Back-propagation learning in expert networks. *IEEE Transactions on Neural Networks*, 3:62–72.

Le Cun, Y., Denker, J., & Solla, S. (1989). Optimal brain damage. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 598–605), San Mateo, CA. Morgan Kaufmann.

LeCun, Y. & Bengio, Y. (1995). Pattern recognition. In Arbib, M., editor, *The Handbook of Brain Theory and Neural Network*, (pp. 711–715), Cambridge, MA. MIT Press.

Liepins, G. & Vose, M. (1990). Representational issues in genetic optimization. *Journal of Experimental and Theoretical Artificial Intelligence*, 2:101–115.

Lincoln, W. & Skrzypek, J. (1989). Synergy of clustering multiple back propagation networks. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 650–659), San Mateo, CA. Morgan Kaufmann.

Litzkow, M., Livny, M., & Mutka, M. (1988). Condor — a hunter of idle workstations. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, (pp. 104–111), San Jose, CA.

MacKay, D. (1992). A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4:448–472.

Maclin, R. & Shavlik, J. (1993). Using knowledge-based neural networks to improve algorithms: Refining the Chou-Fasman algorithm for protein folding. *Machine Learning*, 11:195–215.

Maclin, R. & Shavlik, J. (1994). Incorporating advice into agents that learn from reinforcements. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, (pp. 694–699), Seattle, WA. AAAI/MIT Press.

Maclin, R. & Shavlik, J. (1995). Combining the predictions of multiple classifiers: Using competitive learning to initialize neural networks. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Canada.

Mahoney, J. & Mooney, R. (1993). Combining connectionist and symbolic learning to refine certainty-factor rule-bases. *Connection Science*, 5:339–364.

Mahoney, J. & Mooney, R. (1994). Comparing methods for refining certainty-factor rule-bases. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 173–180), New Brunswick, NJ.

Mangasarian, O. & Solodov, M. (1994). Backpropagation convergence via deterministic nonmonotone perturbed minimization. In Cowan, J., Tesauro, G., & Alspector, J., editors, *Advances in Neural Information Processing Systems (volume 6)*, (pp. 383–390), San Mateo, CA. Morgan Kaufmann.

Mani, G. (1991). Lowering variance of decisions by using artificial neural network portfolios. *Neural Computation*, 3:484–486.

Masuoka, R., Watanabe, N., Kawamura, A., Owada, Y., & Asakawa, K. (1990). Neurofuzzy system — fuzzy inference using a structured neural network. In *Proceedings of the International Conference on Fuzzy Logic & Neural Networks*, (pp. 173–177), Iizuka, Japan.

Mezard, M. & Nadal, J.-P. (1989). Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A*, 22:2191–2204.

Michalski, R. (1983). A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–161.

Miller, G., Todd, P., & Hegde, S. (1989). Designing neural networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, (pp. 379–384), Arlington, VA.

Mitchell, T. (1982). Generalization as search. *Artificial Intelligence*, 18:203–226.

Montana, D. & Davis, L. (1989). Training feedforward networks using genetic algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, (pp. 762–767), Detroit, MI.

Moody, J. (1991). The *effective* number of parameters: An analysis of generalization and regularization in nonlinear learning systems. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 4)*, (pp. 847–854), San Mateo, CA. Morgan Kaufmann.

Mooney, R., Shavlik, J., Towell, G., & Gove, A. (1989). An experimental comparison of symbolic and connectionist learning algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, (pp. 775–780), Detroit, MI.

Mozer, M. C. & Smolensky, P. (1989). Using relevance to reduce network size automatically. *Connection Science*, 1:3–16.

Nowlan, S. & Hinton, G. (1992). Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4:473–493.

Nowlan, S. & Sejnowski, T. (1992). Filter selection model for generating visual motion signals. In Hanson, S., Cowan, J., & Giles, C., editors, *Advances in Neural Information Processing Systems (volume 5)*, (pp. 369–376), San Mateo, CA. Morgan Kaufmann.

Oliker, S., Furst, M., & Maimon, O. (1992). A distributed genetic algorithm for neural network design and training. *Complex Systems*, 6:459–477.

Oliver, W. & Schneider, W. (1988). Using rules and task division to augment connectionist learning. In *Proceedings of the Tenth Annual Conference of the Cognitive Science Society*, (pp. 55–61), Montreal, Canada.

Omlin, C. & Giles, C. (1992). Training second-order recurrent neural networks using hints. In *Proceedings of the Ninth International Conference on Machine Learning*, (pp. 361–366), Aberdeen, Scotland.

Opitz, D. & Shavlik, J. (1993). Heuristically expanding knowledge-based neural networks. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, (pp. 1360–1365), Chambery, France.

Opitz, D. & Shavlik, J. (1994a). Genetically refining topologies of knowledge-based neural networks. In *International Symposium on Integrating Knowledge and Neural Heuristics*, (pp. 57–66), Pensacola, FL.

Opitz, D. & Shavlik, J. (1994b). Using genetic search to refine knowledge-based neural networks. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 208–216), New Brunswick, NJ.

Opitz, D. & Shavlik, J. (1995a). Dynamically adding symbolically meaningful nodes to knowledge-based neural networks. *Knowledge-Based Systems*.

Opitz, D. & Shavlik, J. (1995b). Using heuristic search to expand knowledge-based neural networks. In Petsche, T., Hanson, S., & Shavlik, J., editors, *Computational Learning Theory and Natural Learning Systems (volume 3)*. MIT Press, Cambridge, MA.

Ourston, D. & Mooney, R. (1994). Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66:273–309.

Pazzani, M. & Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning*, 9:57–94.

Perrone, M. (1992). A soft-competitive splitting rule for adaptive tree-structured neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, (pp. 689–693), Baltimore, MD.

Pomerleau, D. (1991). Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3:88–97.

Provost, F. & Danyluk, A. (1995). Learning from bad data. In *Workshop on Applying Machine Learning in Practice*, held at *the Twelfth International Conference on Machine Learning*, Tahoe City, CA.

Quinlan, J. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

Quinlan, J. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA.

Quinlan, J. & Cameron-Jones, R. (1995). Lookahead and pathology in decision tree induction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Canada.

Rabinowitz, H., Flamholz, J., Wolin, E., & Euchner, J. (1991). NYNEX MAX: A telephone trouble screening expert. In *Innovative Applications of Artificial Intelligence 3*, (pp. 213–230), Menlo Park, CA.

Rich, E. (1983). *Artificial Intelligence*. McGraw-Hill, New York.

Richards, B. (1995). Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19:95–131.

Roscheisen, M., Hofmann, R., & Tresp, V. (1991). Neural control for rolling mills: Incorporating domain theories to overcome data deficiency. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 4)*, (pp. 659–666), San Mateo, CA. Morgan Kaufmann.

Rost, B. & Sander, C. (1993). Prediction of protein secondary structure at better than 70% accuracy. *Journal of Molecular Biology*, 232:584–599.

Rumelhart, D., Durbin, D., Golden, R., & Chauvin, Y. (1995). Backpropagation: The basic theory. In Chauvin, W. & Rumelhart, D., editors, *Backpropagation: Theory, Architectures, and Applications*, (pp. 1–34), Hillsdale, NJ. Lawrence Erlbaum Associates.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning internal representations by error propagation. In Rumelhart, D. & McClelland, J., editors, *Parallel Distributed Processing: Explorations in the microstructure of cognition. Volume 1: Foundations.* MIT Press, Cambridge, MA.

Schiffmann, W., Joost, M., & Werner, R. (1992). *Synthesis and performance analysis of multilayer neural network architectures.* Technical Report 16, University of Koblenz, Institute for Physics.

Scott, G., Shavlik, J., & Ray, W. (1992). Refining PID controllers using neural networks. *Neural Computation*, 5:746–757.

Sejnowski, T. & Rosenberg, C. (1987). Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145–168.

Sestito, S. & Dillon, T. (1990). Using multi-layered neural networks for learning symbolic knowledge. In *Proceedings of the 1990 Australian Artificial Intelligence Conference*, Perth, Australia.

Shapire, R. (1990). The strength of weak learnability. *Machine Learning*, 5:197–227.

Shavlik, J. (1994). Combining symbolic and neural learning. *Machine Learning*, 14:321–331.

Shavlik, J. & Dietterich, T., editors (1990). *Readings in Machine Learning.* Morgan Kaufmann, San Mateo, CA.

Shavlik, J. & Towell, G. (1989). An approach to combining explanation-based and neural learning algorithms. *Connection Science*, 1:233–255.

Stone, M. (1974). Cross-validation choice and assessment of statistical predictions. *Journal of the Royal Statistical Society B*, 36:111–147.

Stormo, G. (1987). Identifying coding sequences. In Bishop, M. J. & Rawlings, C. J., editors, *Nucleic Acid and Protein Sequence Analysis: A Practical Approach.* IRL Press, Oxford, England.

Suzuki, D., Griffiths, A., Miller, J., & Lewontin, R. (1989). *An Introduction to Genetic Analysis.* W. H. Freeman and Company, New York, fourth edition.

Tishby, N., Levin, E., & Solla, S. (1989). Consistent inference on probabilities in layered networks, predictions and generalization. In *International Joint Conference on Neural Networks*, (pp. 403–410), Washington, D.C.

Towell, G. (1991). *Symbolic Knowledge and Neural Networks: Insertion, Refinement, and Extraction*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI.

Towell, G. & Shavlik, J. (1992). Using symbolic learning to improve knowledge-based neural networks. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, (pp. 177–182), San Jose, CA.

Towell, G. & Shavlik, J. (1993). Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13:71–101.

Towell, G. & Shavlik, J. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence*, 70:119–165.

Tresp, V., Hollatz, J., & Ahmad, S. (1992). Network structuring and training using rule-based knowledge. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 5)*, (pp. 871–878), San Mateo, CA. Morgan Kaufmann.

Tresp, V. & Taniguchi, M. (1995). Combining estimators using non-constant weighting functions. In Tesauro, G., Touretzky, D., & Leen, T., editors, *Advances in Neural Information Processing Systems (volume 7)*, Cambridge, MA. MIT Press.

Tsoi, A. & Pearson, R. (1990). Comparison of three classification techniques, CART, C4.5, and multi-layer perceptrons. In Lippmann, R., Moody, J., & Touretzky, D., editors, *Advances in Neural Information Processing Systems (volume 3)*, (pp. 963–969), San Mateo, CA. Morgan Kaufmann.

Turney, P. (1995). Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm. *Journal of Artificial Intelligence Research*, 2:369–409.

Uberbacher, E. C. & Mural, R. J. (1991). Locating protein coding regions in human DNA sequences using a neural network – multiple sensor approach. *Proceedings of the National Academy of Sciences (USA)*, 88:11261–11265.

Wan, E. (1993). Time series prediction using a connectionist network with internal delay lines. In Weigend, A. & Gershenfeld, N., editors, *Time Series Prediction: Forecasting the Future and Understanding the Past*, (pp. 195–217), Reading, MA. Addison-Wesley.

Watrous, R., Towell, G., & Glassman, M. (1995). Synthesize, optimize, analyze, repeat (SOAR): Application of neural network tools to ECG patient monitoring. In *Proceedings of the Workshop on Environmental and Energy Applications of Neural Networks*, Richland, WA.

Watson, J. (1990). The Human Genome Project: Past, present, and future. *Science*, 248:44–48.

Watson, J. D., Hopkins, N. H., Roberts, J. W., Argetsinger Steitz, J., & Weiner, A. M. (1987). *Molecular Biology of the Gene*. Benjamin/Cummings, Menlo Park, CA, fourth edition.

Weigend, A. (1993). On overfitting and the effective number of hidden units. In *Proceedings of the 1993 Connectionist Models Summer School*, (pp. 335–342), Boulder, CO.

Weigend, A. & Gershenfeld, N. (1992). *Time Series Prediction: Forecasting the Future and Understanding the Past*. Addison-Wesley, Reading, MA.

Weigend, A., Huberman, B., & Rumelhart, D. (1990). Predicting the future: A connectionist approach. *International Journal of Neural Systems*, I:193–209.

Weiss, S. M. & Kulikowski, C. A. (1990). *Computer Systems that Learn*. Morgan Kaufmann, San Mateo, CA.

Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University.

Whitley, D. & Hanson, T. (1989). Optimizing neural networks using faster, more accurate genetic search. In *Proceedings of the Third International Conference on Genetic Algorithms*, (pp. 391–396), Arlington, VA.

Wilson, S. (1991). GA-easy does not imply steepest-ascent optimizable. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, (pp. 85–89), San Diego, CA.

Winston, P. (1975). Learning structural descriptions from examples. In Winston, P., editor, *The Psychology of Computer Vision*, (pp. 157–210), New York. McGraw-Hill.

Wolpert, D. (1992). Stacked generalization. *Neural Networks*, 5:241–259.

Wynne-Jones, M. (1991). Node splitting: A constructive algorithm for feed-forward neural networks. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 4)*, (pp. 1072–1079), San Mateo, CA. Morgan Kaufmann.

Yao, X. (1993). Evolutionary artificial neural networks. *International Journal of Neural Systems*, 4:203–221.

Zhang, X., Mesirov, J., & Waltz, D. (1992). Hybrid system for protein secondary structure prediction. *Journal of Molecular Biology*, 225:1049–1063.